AFRL-IF-RS-TR-2004-331
Final Technical Report
December 2004

SLIIC: SYSTEM-LEVEL INTELLIGENT INTENSIVE COMPUTING

USC Information Sciences Institute

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. J200

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

STINFO FINAL REPORT

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2004-331 has been reviewed and is approved for publication

APPROVED: /s/
CHRISTOPHER J. FLYNN
Project Engineer

FOR THE DIRECTOR: /s/
JAMES A. COLLINS, Acting Chief
Information Technology Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 2004	**FINAL** **Mar 99 – Sep 03**

4. TITLE AND SUBTITLE

SLIIC: SYSTEM-LEVEL INTELLIGENT INTENSIVE COMPUTING

5. FUNDING NUMBERS

G - F30602-99-1-0521
PE - 62301E
PR - H306
TA - SL
WU - IC

6. AUTHOR(S)

Stephen Crago
Jinwoo Suh

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

USC Information Sciences Institute
4676 Admiralty Way
Marina Del Rey CA 90292-6695

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington VA 22203-1714

AFRL/IFTC
525 Brooks Road
Rome NY 13441-4505

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

AFRL-IF-RS-TR-2004-331

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Christopher J. Flynn/IFTC/(315) 330-3249

12a. DISTRIBUTION / AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 Words)***

Recently, several architectural approaches have been explored that promise to hide memory latency for applications that include data-intensive applications while improving scalability The SLIIC project demonstrated and compared some of the advantages and disadvantages of the PIM (processor-in-memory) and stream processing approaches to hiding memory latency. The SLIIC project built board prototypes for PIM and stream processing architectures and implemented data-intensive applications in simulation and in hardware to measure the performance. Speedups of up to 54 measured in cycles and 16 measured in execution time were obtained over commercial microprocessors.

14. SUBJECT TERMS

Data-intensive computing architectures, processor-in-memory, stream processing

15. NUMBER OF PAGES 115

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Table of Contents

Figures

Tables

I. INTRODUCTION

Microprocessor performance has been doubling every 18-24 months for many years [5]. This performance increase has been possible because die size has increased and feature size has decreased. Unfortunately, main memory latency has not been improving at the same rate, resulting in a growing gap between microprocessor speed and memory speed. Latency of DRAM (dynamic random access memory), the technology used for main memory, has only improved by 7% per year [5], and memory bandwidth, limited by pin bandwidth has also failed to keep pace. These growing gaps have created a problem for data-intensive applications.

Increasing die size combined with fast clock speeds has made the maximum distance as measured in clock cycles between two points on a processor longer. Pipelining has been widely used to address this problem. However, increasing pipeline depth increases various latencies, including cache access and branch prediction penalties, and increases the complexity of processor design. Other techniques for exploiting ILP (instruction level parallelism) without exposing parallelism to the instruction set, such as superscalar out-of-order processing, have also reached a point of diminishing returns.

To bridge these growing gaps, many methods have been proposed such as caching, prefetching, and multithreading. These methods, however, provide limited performance improvement and can even hinder performance for data-intensive applications. Caching has been the most popular technique [13][15]. It increases performance by utilizing temporal and spatial locality, but it is not useful for many data-intensive applications since many of them do not show such locality [14].

Recently, several architectural approaches have been explored that promise to hide memory latency for applications that include data-intensive applications while improving scalability. The SLIIC project demonstrated and compared some of the advantages and disadvantages of the PIM (processor-in-memory) and stream processing approaches to the problems described above. The SLIIC project built board prototypes for PIM and stream processing architectures and implemented data-intensive applications in simulation and in hardware to measure the performance.

The rest of this report is organized as follows. Chapter II describes two PIMs, a stream processor, and a tile-based processor that were used in the SLIIC project. Chapter III describes the four kernels we implemented: the corner turn, coherent side-lobe canceller, beam steering, and digital target generator. Also, the techniques used to exploit each architecture are described. In Chapter IV, the implementation results and analysis are shown. Chapter V concludes the report.

II. DATA-INTENSIVE PROTOTYPE ARCHITECTURES

In this section, the M32R/D, V-IRAM (Vector Intelligent Random Access Memory), Imagine, and Raw chips and their boards are described. Note that even though Raw was not developed under SLIIC or DARPA's Data Intensive Systems program, it is included here for comparison. We also describe the performance models that will be used to understand performance of the kernels.

II.A. The Mitsubishi M32R/D PIM

In conventional systems, the CPU (Central Processing Unit) and memory are implemented on different chips. Thus, the bandwidth between CPU and memory is limited since the data must be transferred through chip I/O pins and copper wires on a printed circuit board. Furthermore, much of the internal structure of DRAM, which could be exploited if exposed, is hidden because of the bandwidth limitation imposed by the pins.

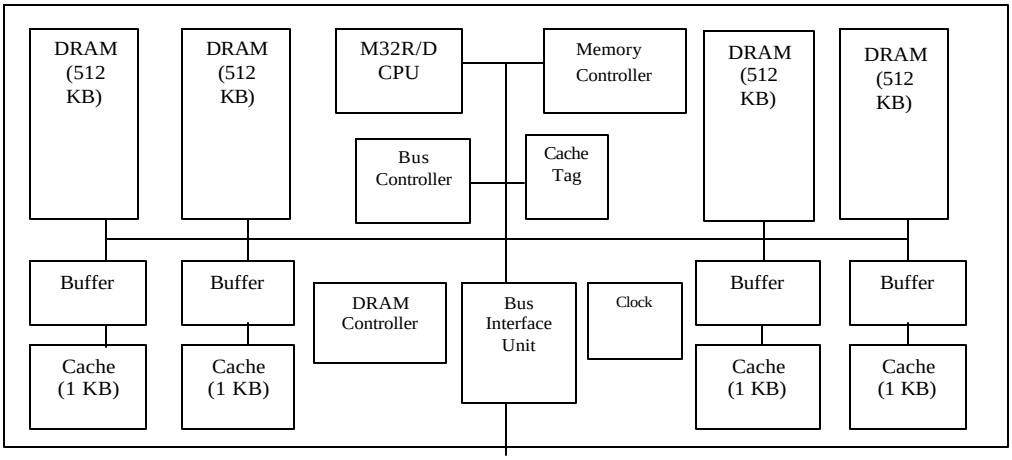


Figure 1. Block diagram of M32R/D

Processor-In-Memory (PIM) technology is a method for closing the gap between memory speed and processor speed for data intensive applications. PIM technology integrates a processor and DRAM on the same chip. The integration of memory and processor on the same chip has the potential to decrease memory latency and increase the bandwidth between the processor and memory. PIM technology also has the potential to decrease other important system parameters such as power consumption, cost, and area.

The SLIIC project chose to use the Mitsubishi M32R/D chip [9] to determine baseline performance for a PIM that could be compared to a traditional microprocessor. At the time, the M32R/D was the only commercial general-purpose PIM chip that had more than 1 MB of DRAM. The M32R/D processor was chosen because it provides a relatively large memory size (2 MB) and high data path width between memory and cache (128 bits). It also has a small footprint that enables many processors to fit on a board. A block diagram of the M32R/D is shown in Figure 1. It contains a 32-bit RISC (Reduced Instruction Set Computer) CPU and 2-megabytes of internal DRAM. Between the CPU and DRAM, there is a 4-kilobyte cache. The bus width between the cache and DRAM is 128 bits. 128 bits of data can be transferred between cache and memory in a single clock cycle; however, since the CPU is 32 bits wide, only 32 bits

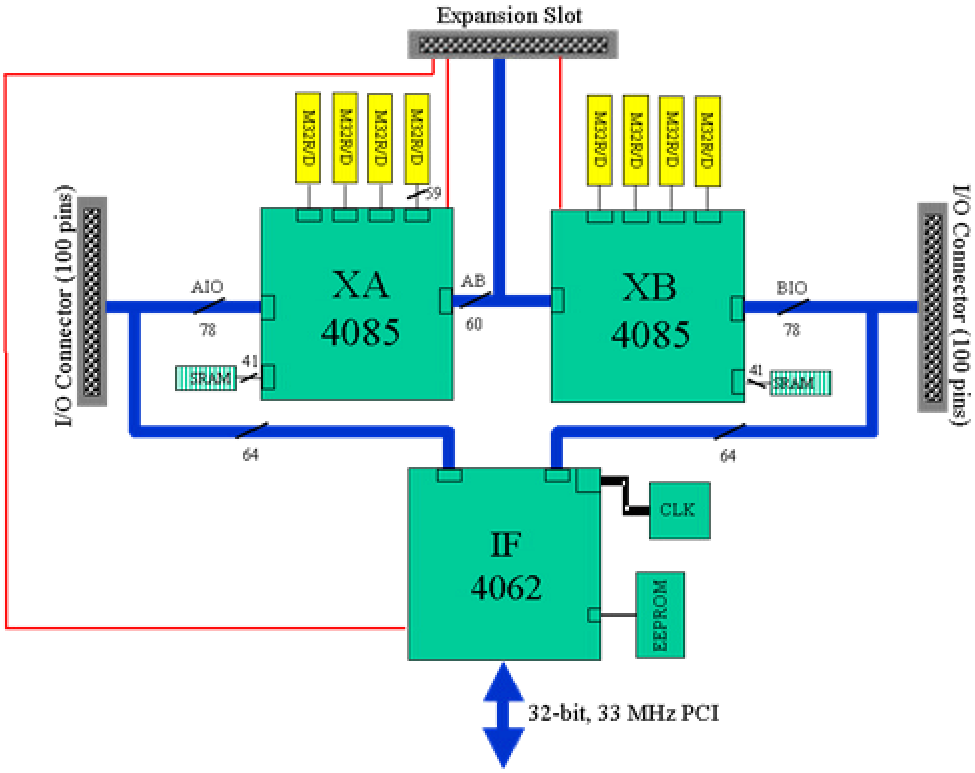


Figure 2. Block diagram of SLIIC-QL board

of data can be transferred between the CPU and cache in a clock cycle. The clock speed of the external bus interface is 20 MHz, and the internal clock speed is 80 MHz. The processor does not support floating-point operations directly in hardware, so floating point operations are performed in software. We used integer versions of benchmark kernels so that we could measure performance characteristics of the PIM interface, rather than the performance limitations of the software-implemented floating-point operations. The SLIIC project built a board called the SLIIC Quick Look (QL) board to demonstrate the density that could be achieved with PIM components and to measure the performance of our benchmarking kernels. A block diagram of the SLIIC Quick Look (QL) board is shown in Figure 2 and a photograph of the board is shown in Figure 3. The board fits in a standard PCI (Peripheral Computer Interface) form factor, allowing it to be plugged into a host PC (Personal Computer) platform. It contains eight Mitsubishi M32R/Ds, providing 640 MIPS (Million Instructions Per Second) of peak processing power and 16 MB of memory.

The XA and XB components shown in Figure 2 are FPGAs (field programmable gate arrays) that serve several purposes. First, the FPGAs provide a multiprocessor interconnect for the M32R/Ds. Second, XA and XB provide programmable logic that can be used for processor synchronization and performance measurement. Each FPGA has an attached SRAM (Synchronous Random Access Memory) memory that can be used to store tables of performance counters and data. Third, the FPGAs provide logic that facilitates communication with the host PC.

There are two advantages to implementing these functions in FPGAs. First, modification of the functions and circuits can be done without changing the hardware. The FPGA

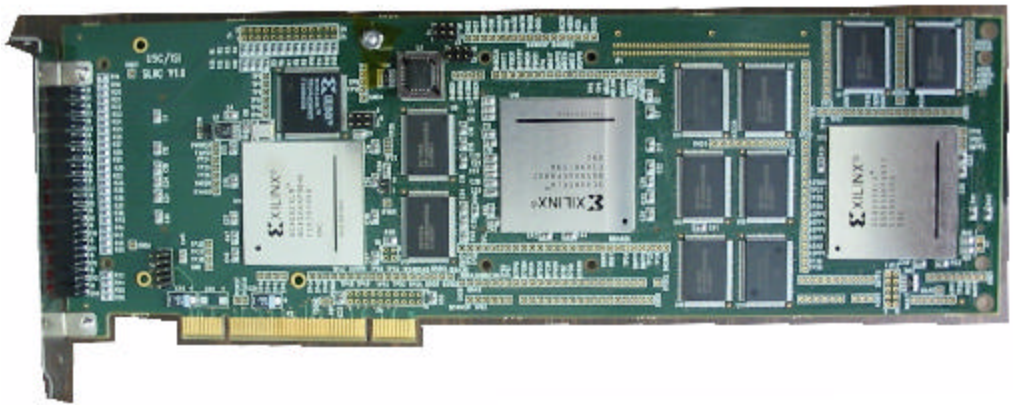


Figure 3. SLIIC-QL board

configurations are specified using the VHDL hardware description language. This is critical for research and initial prototypes, where many changes are inevitable. Second, the flexibility allows the board to be used for other applications. When the board is used for another application, the logic in the FPGAs is changed to accommodate new needs.

The two interconnection FPGAs implement logic as shown in Figure 4, where P0 through P7 represent the M32R/D chips. In each FPGA, there is a 7x7 pipelined crossbar switch. Four crossbar inputs and outputs are connected to the four processors in the cluster (where a cluster means four processors connected to an FPGA). When communication is performed among processors in a cluster, the 4x4 sub-crossbar is used. To communicate with a processor in a different cluster, one of the two paths between the two clusters must be used. To use a path, the processors use one of the two ports connected to the two paths in the 7x7 crossbar. The remaining port (omitted from Figure 4) connects to the interface FPGA (IF), through which PIMs communicate to host PC. The crossbar configuration does not allow a processor to receive data from more than one processor simultaneously. The application programmer is responsible for avoiding communication conflicts since no hardware or firmware prevention of conflicts has been provided. This enables a simpler design, and, as a result, the network is faster.

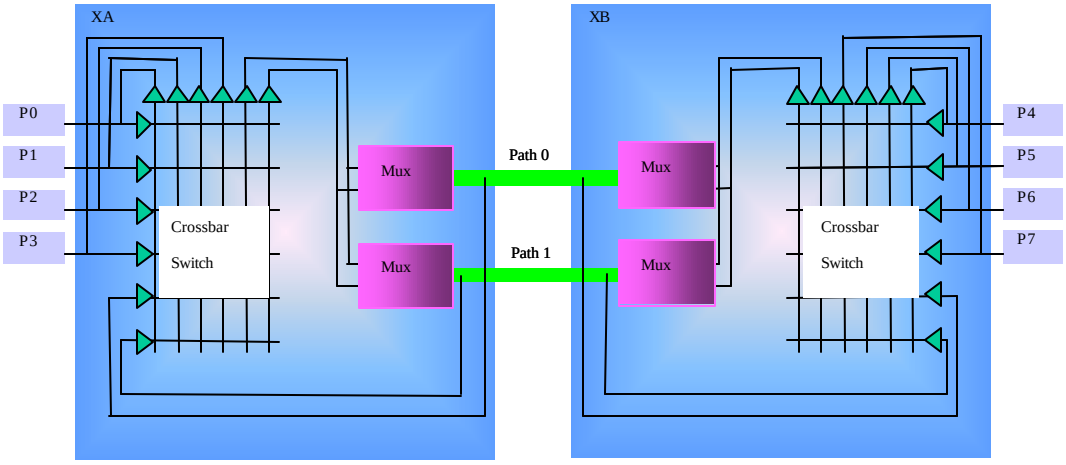


Figure 4. Interconnection network in FPGAs on SLIIC-QL board

Communication between PIM chips is performed through a hybrid message-passing/shared-memory scheme. Each processor has access to internal memory (addresses 0 to 0x1F FFFF) and to the memory of remote PIMS (addresses 0x20 0000 to 0x3F FFFF). All remote memory (memory on other PIM chips) is mapped to the same address space. A destination identification number sent before a message determines the actual PIM that is mapped to the external memory address range at any given time. For intra-cluster communication, the crossbar pipeline has three stages.

The two clusters communicate through two half-duplex bi-directional paths. The application programmer or system software needs to set the path number to use in addition to the destination processor ID before sending a message to prevent resource conflicts. Inter-cluster communication has five pipeline stages. For processor synchronization, barrier synchronization is supported using FPGA logic. Barrier synchronization can be performed with overhead of just a few system clock cycles.

The SLIIC QL board communicates with the host PC using the SLIIC debugger. The SLIIC debugger, which runs on Windows NT, allows a user to start execution, read data from PIMs, write data to PIMs, and read counter data (there is a counter associated with each processor that is used for measurement of execution time). On start-up, the debugger initializes itself and automatically sets the SLIIC QL board clock speed to 20 MHz. For application programmers, an API (Application Programming Interface) is provided that includes a communication interface, barrier synchronization, and timer control.

II.B. The V-IRAM Processor

The V-IRAM chip is a PIM research prototype developed at the University of California at Berkeley [7]. The architecture of the chip is shown in Figure 5. The V-IRAM contains two vector-processing units (ALU0 and ALU1) in addition to a scalar-processing unit. These units are pipelined. The vector functional units can be partitioned into several smaller units, depending on the arithmetic precision required. For example, a vector functional unit can be partitioned into 4 units for 64-bit operations or 8 units for 32-bit operations. Some operations are allowed to execute on ALU0 only. It has an 8K vector register file with 32 addressable registers.

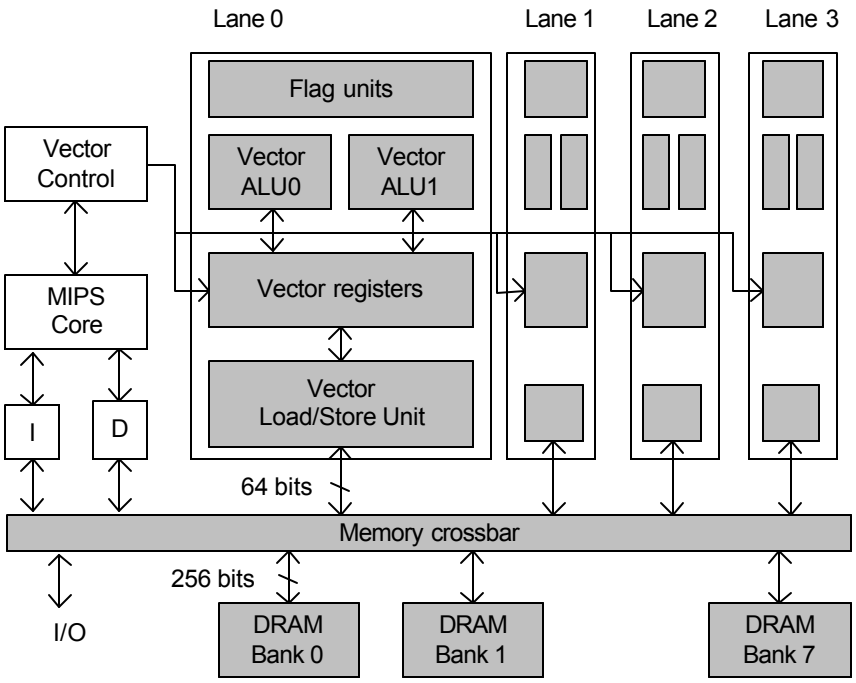


Figure 5. Block diagram of V-IRAM

The V-IRAM has 13 MB of DRAM. There is a 256-bit data path between the processing units and DRAM. The DRAM is partitioned into two wings, each of which has four banks. It can access eight sequential 32-bit data elements per clock cycle; however, since there are four address generators, it can access only four strided 32-bit or 64-bit data elements per cycle.

There is a crossbar switch between the DRAM and the vector processor. The target processor speed is 200 MHz, which would provide a peak performance of 3.2 GOPS (200 MHz x 2 functional units x 8 data elements per clock) for 32-bit data. If 16-bit data is processed, the performance is 6.4 GOPS. Its peak floating point performance is 1.6 GFLOPS for 32-bit floating-point operations. The power consumption is expected to be about 2 W. The EEMBC (Embedded Microprocessor Benchmark Consortium) benchmarks have been implemented on V-IRAM. V-IRAM's performance is 10 to 100 times better (as normalized by clock frequency) than the MPC 7455 processor depending on memory access pattern and how vectorizable the code is [9].

A block diagram of the V-IRAM prototype that was developed as part of the SLIIC project is shown in Figure 6. The prototype was implemented as two boards: one board with the interface logic and one board with the V-IRAM chip that mates to the glue logic board. This

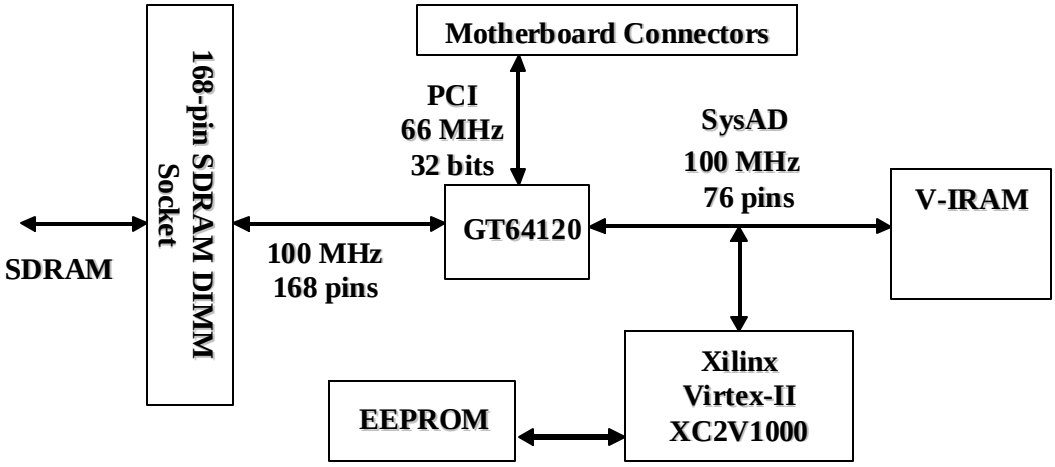
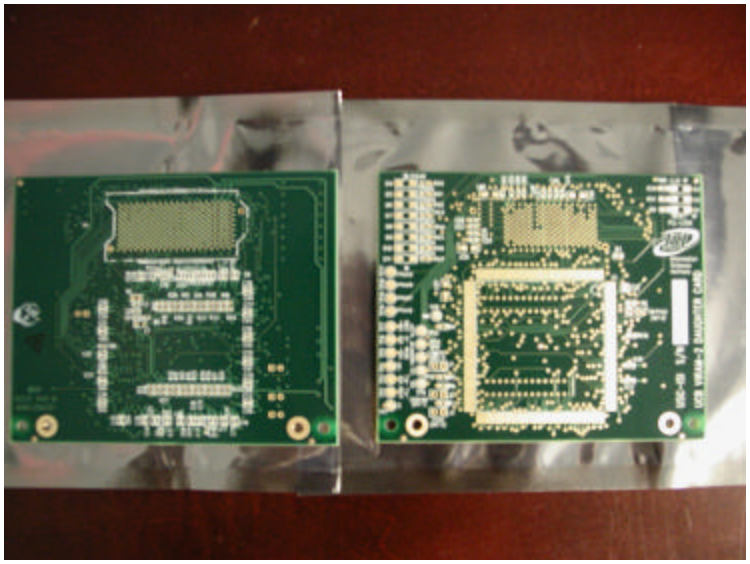


Figure 6. Block diagram of V-IRAM board

implementation allows individual non-functional V-IRAM chips to be discarded while re-using the glue logic. The assembly described here plugs into a commercial MIPS testing board that provides peripherals and software infrastructure that can be re-used for V-IRAM. Photos of the boards are shown in Figure 7. The board has been fabricated and was delivered for testing by the V-IRAM group at Berkeley. Testing has not been completed at this time because a re-spin of the V-IRAM chip is being done to correct a design error in the V-IRAM chip.

II.C. The Imagine Stream Processor

Another approach for handling the growing processor-memory gap is stream processing. In this approach, the data is routed through stream registers to hide memory latency, allow the re-ordering of DRAM accesses, and minimize the number of accesses to external memory components. The Imagine chip is a research prototype stream processor developed at Stanford University [6][8][12][14]. It contains eight clusters of arithmetic units that process data from a stream register file. The processor speed is currently 300 MHz, which provides a peak performance of over 14 GOPS (32-bit integer or floating-point operations). Performance results for Imagine have been presented for application kernels such as MPEG and QRD [12]. Arithmetic logic unit (ALU) utilization between 84% and 95% is reported for streaming media applications.



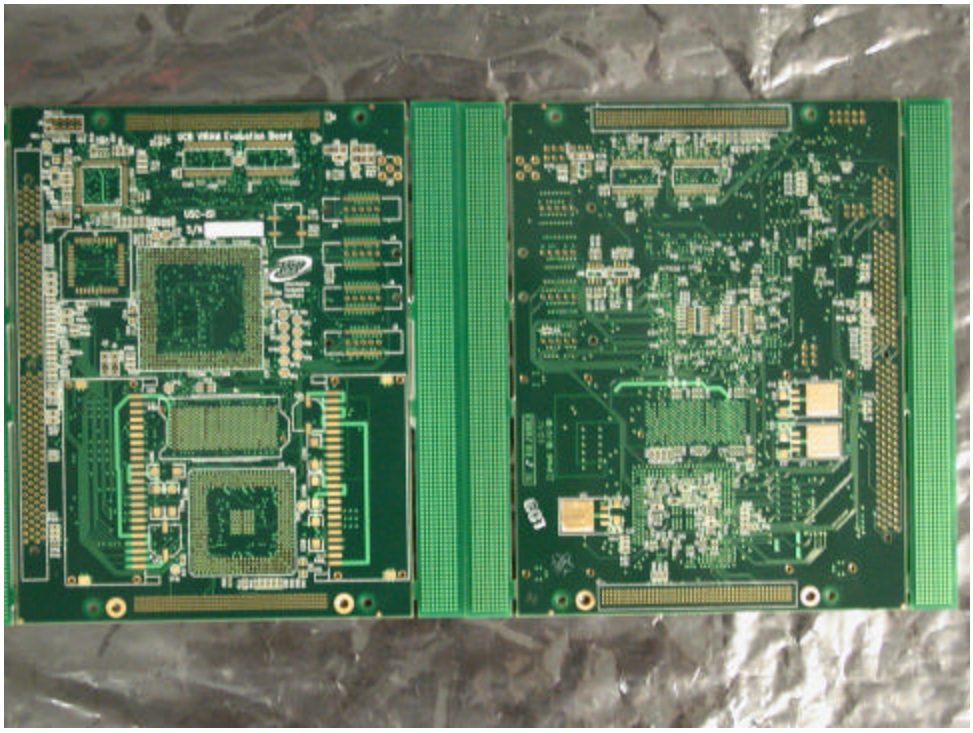


Figure 7. Photos of V-IRAM daughter card

Figure 8 shows a block diagram of Imagine. The stream processing is implemented with eight ALU clusters (with 6 ALUs each), a large stream register file (SRF), and a high-bandwidth interconnect between them. The eight ALU clusters operate on data from the SRF. The size of the SRF is 128 kilobytes. Up to eight input or output streams can be processed simultaneously. The data is sent to clusters in round-robin fashion, i.e., the *i*-th data is sent to cluster (*i* mod 8). All clusters perform the same operations on their data in SIMD (Single Instruction, Multiple Data) style. Each cluster has 6 arithmetic units (three adders, two multipliers, and one divider) and one communication interface that is used to send data between ALU clusters. A stream can

start at the start of any SRF 128-byte block. Data is transferred to and from the SRF from off-chip memory or the network interface. The Imagine chip prototype implementation has two memory controllers, each of which can process a memory access stream. The memory controller reorders accesses to reduce bank conflicts and to increase data access locality.

The first generation Imagine multiprocessor board, called the Dual Imagine Board, was developed at USC/ISI in collaboration with Stanford. The block diagram is shown in Figure 9

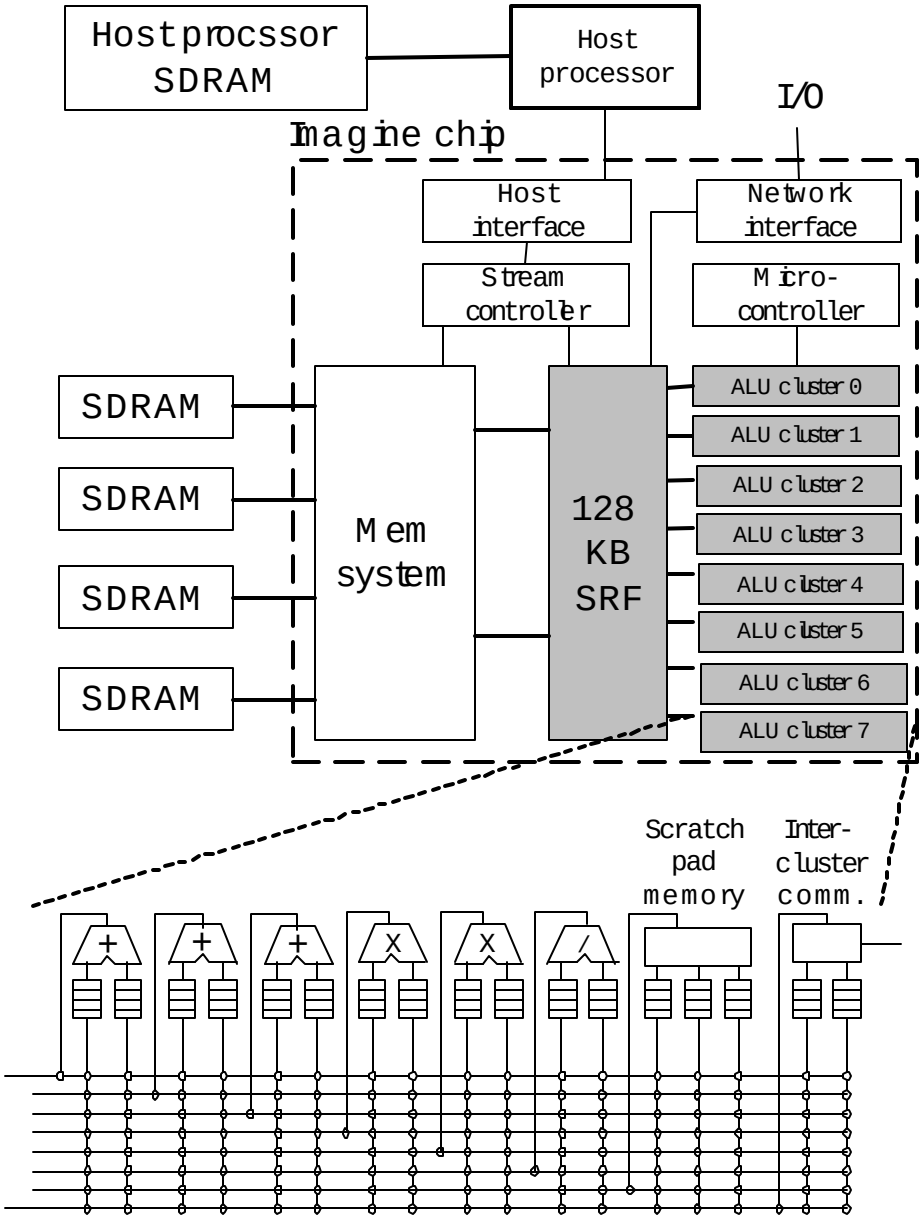


Figure 8. Block diagram of Imagine

and photo of the board is shown in Figure 10. The board contains two Imagine chips, each of which is connected to local SDRAM (Synchronous DRAM). Both Imagines are connected to a

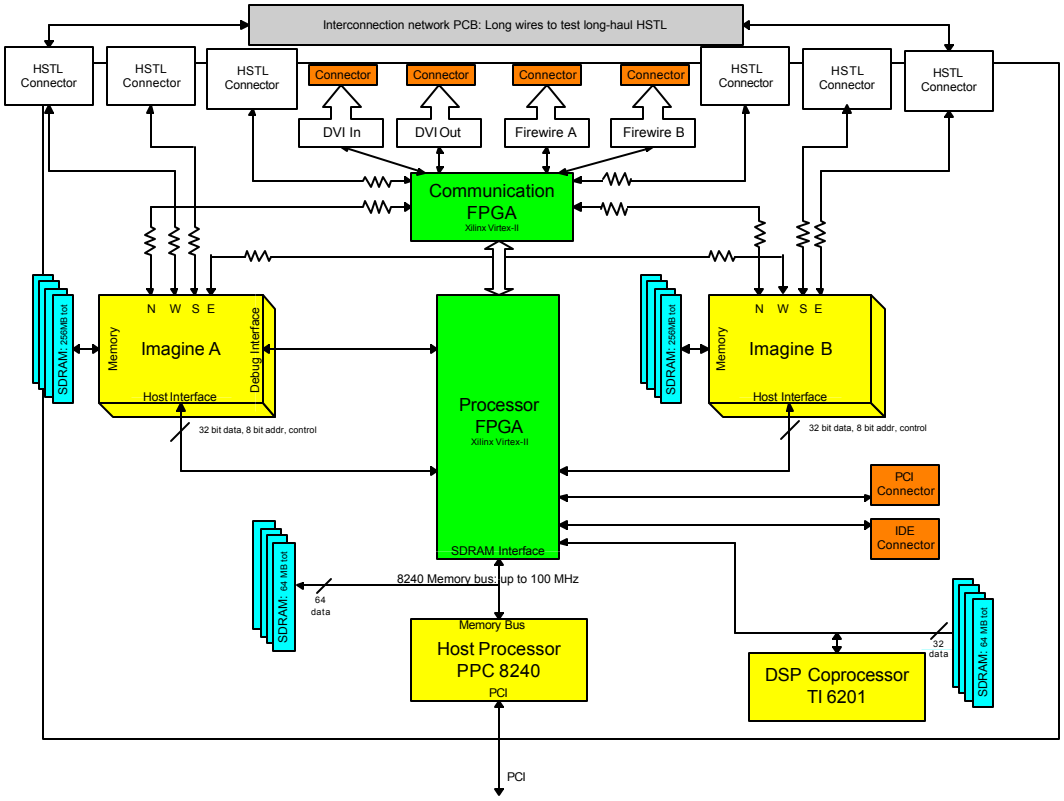


Figure 9. Block diagram of Imagine board

PowerPC host processor through an FPGA chip. The FPGA provides a means of connection between the PowerPC and Imagine by emulating an SDRAM interface on the PowerPC side and an SRAM interface on the Imagine side. It also provides a connection to the DSP (digital signal processor) chip. The PowerPC has its own local SDRAM memory. The PowerPC processor communicates with a host PC through a PCI bus. Applications are compiled on the host PC and sent to the PowerPC through the PCI bus. The PowerPC performs address translation for the PCI memory space, so data written on the PCI memory space by the host PC is actually written to the local memory of the PowerPC. During execution of an application, when the PowerPC encounters kernel code that needs to be executed by an Imagine chip, the PowerPC sends instructions to the Imagine. The Imagine performs the computation and returns the data back to the PowerPC. The board also supports digital video, Firewire, and expansion ports, which are connected to the Imagine chips through another FPGA. The Imagine chips are mounted to the board on a socket to allow the identification of functional Imagine chips.

The overall Imagine system is shown in Figure 11. The debugger on the host PC accepts user commands and interprets them. Data and PowerPC instructions are sent to the PowerPC through the PCI bus. The PowerPC executes control code and sends kernels and their data to the Imagine chip. The code for the Imagine chip is generated by the Imagine compiler and the Imagine kernel scheduler, which compile code written in StreamC and KernelC.

Figure 10. Photograph of the Dual-Imagine Board

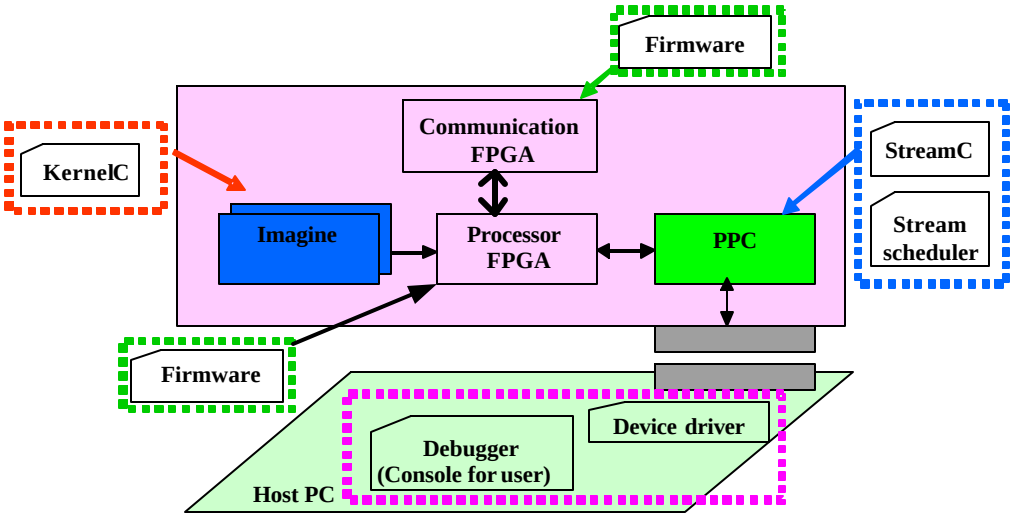


Figure 11. Block diagram of Dual Imagine system

A second generation Imagine board is being developed at ISI in conjunction with Stanford. The new board has a similar architecture, but has a PowerPC core integrated into the interface FPGA associated with each Imagine chip. The second key change is that the Imagine chips will be directly soldered to the board, which will allow more reliable and faster connections. The sockets on the original Dual Imagine board have been used for tests designed to identify good Imagine chips that will populate the revised boards. A picture of an unpopulated board is shown in Figure 13.

II.D. The Raw Tiled Architecture

Although any efforts related to Raw have not been performed under SLIIC, some results are included here for comparison. An approach for a scalable microprocessor that addresses

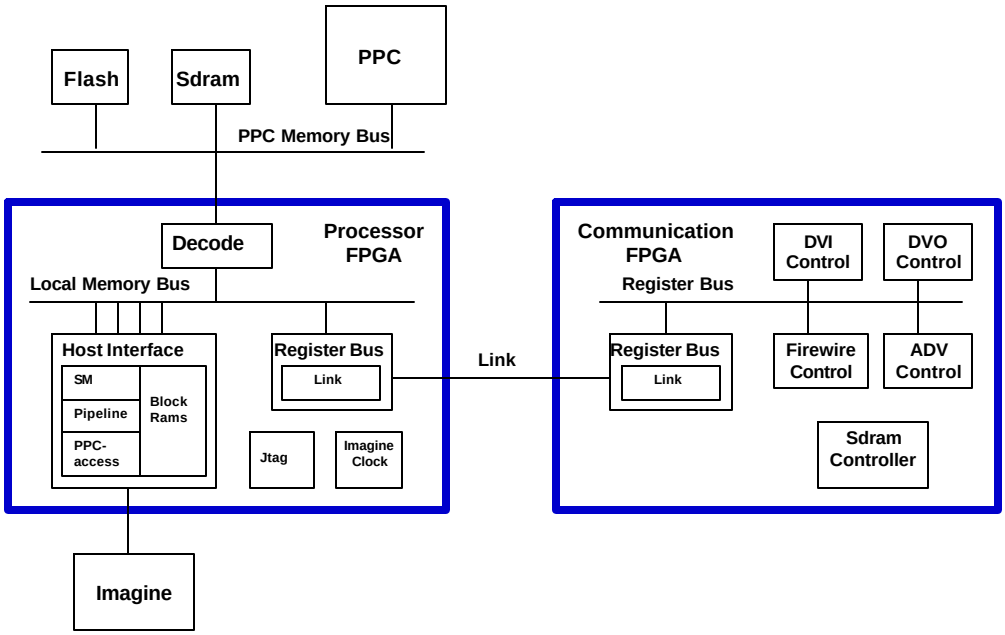


Figure 12. Block diagram of Dual Imagine Board FPGAs

issues of continued technology scaling is tile processing. Instead of building one processor on a chip, several processors (tiles) are implemented and connected in a mesh topology. The tiles enable faster clock frequencies since the signals need to travel only a short distance and allows performance to scale for applications that can use multiple tiles effectively. An example of a tiled architecture is the Raw chip implemented at MIT [19] and shown in Figure 14. The current Raw implementation contains 16 tiles on a chip connected by a very low latency 2-D mesh scalar operand network [20]. The Raw chip prototype has been tested at up to 400MHz; however, interface logic on the Raw board limits the clock frequency to 300 MHz. Peak performance is 4.8 GOPS and 6.4 GOPS at 300 MHz and 400 MHz, respectively.

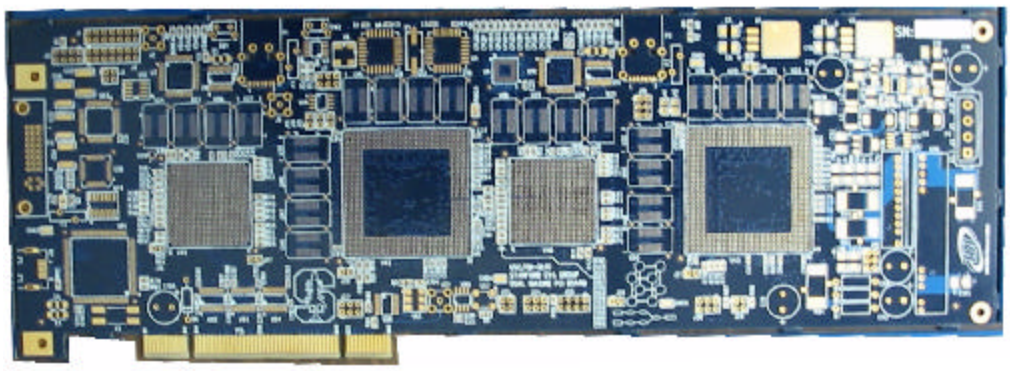


Figure 13. Photograph of second generation Dual Imagine Board

Each tile has a MIPS-based RISC processor with floating-point units and a total of 128 kilobytes of SRAM, which includes switch instruction memory, tile (processor) instruction memory, and data memory. Raw can exploit streaming, instruction-level, MIMD (Multiple Instruction, Multiple Data), and data parallelism.

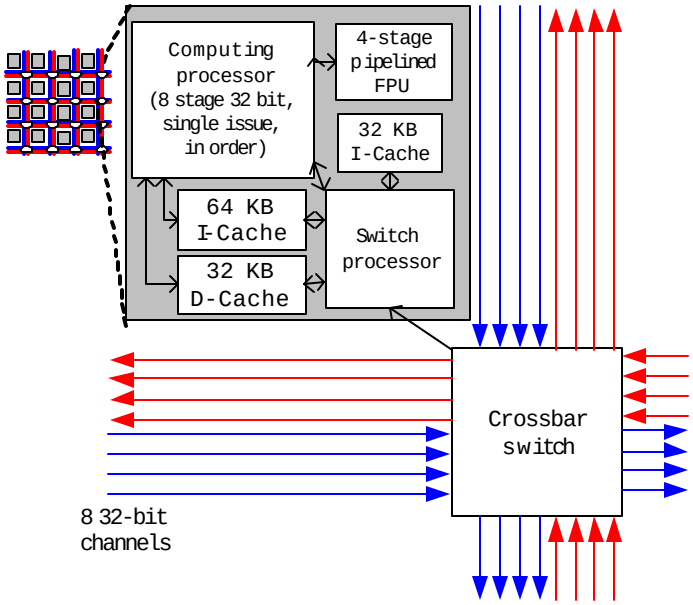


Figure 14. Block diagram of a Rawtile

The Raw processor has four networks, two static and two dynamic. Communication on the static networks is routed by a switch processor in each tile [19]. The switch processor is located between the computation processor and the network and provides throughput to the tile processor of one word per cycle with a latency of three cycles between nearest neighbor tiles. One additional cycle of latency is added for each hop in the mesh through the static networks. The dynamic networks are packet-based, and have longer latencies since routing must be determined dynamically in hardware. The memory ports are located at the 16 peripheral ports of the chip. All tiles can access memory either through the dynamic network or through the static network.

Several kernels, including matrix multiplication, are implemented on Raw and the results are reported in [20]. The results show that Raw obtains speedup of up to 12 relative to single-tile performance on ILP benchmarks. Speedups greater than 16 can be achieved on streaming benchmarks when compared to a single-issue load/store RISC architecture because of a tile's ability to operate on data directly from the networks.

II.E. Programming Methodologies

The programming methodologies and tools for each of these architectures are evolving. However, each architecture has inherent properties that affect the programming model and programmability of the architecture.

The M32R/D's programming model is similar to that of a conventional processor for one processor. The internal DRAM is treated as a conventional DRAM. The M32R/D does not explicitly support multiprocessor communication. We used a custom message-based communication API programmed on the M32R/D shared memory hardware to implement multiprocessor communication.

The V-IRAM's programming model is that of a traditional vector architecture. An application is described as single instruction stream that contains scalar and vector instructions. There are two primary difficulties to programming the V-IRAM architecture. First, the C programming language makes automatic parallelization of many loops difficult or impossible without making assumptions about the independence of pointer and array accesses. Simple loops or computations marked by user hints can be vectorized, but kernels with complex access patterns like the Fast Fourier Transform (FFT) are still difficult to automatically vectorize. Languages that are more restricted will facilitate automatic vectorization. The second factor that complicates the programmability of V-IRAM is the impact of the DRAM organization on performance. Much of the performance of V-IRAM is achieved by exploiting properties of DRAM organization (e.g. banks, rows, columns, and wings). Currently, the user must understand the DRAM organization to optimize performance. However, it is feasible that a compiler could organize memory references based on memory organization while it is vectorizing, especially given a language that makes this analysis feasible. For this study, a C compiler was used to compile the kernels, and then inner loops were hand-vectorized using assembly code.

The programming model of Imagine has two significant characteristics. First, the programming model is based on streams. Streams are similar to vectors, but streams can be explicitly routed between the stream register files and the ALU clusters without going through the memory system. This property is important for reducing the impact of the bandwidth bottleneck between DRAM and the processor chip. The second significant characteristic of the current Imagine programming model is that a program is described in two languages, one for the host (or control) thread written in C and one for the stream processing unit written in kernel-C. Again, new programming languages may allow this distinction to be hidden from the programmer. However, the programming model used in this project forces the programmer to think explicitly about streams and their control. This explicit streaming model has the disadvantage that a programmer must think about the application in a new way, but has the advantage that the programmer is forced to think about issues that are important to performance anyway. Applications must contain SIMD parallelism to see significant performance improvements on the Imagine architecture. For this study, inner loops were carefully scheduled to maximize performance.

The Raw architecture is the most flexible of the architectures addressed in this report. The tile-based organization with the low-latency, high-bandwidth network, and memory interface supports a variety of programming models. The primary programming models used in the kernels described in this report are the MIMD and stream models. We report results on two modes of using Raw: an easy-to-program, but less efficient MIMD mode, in which data is routed to local memories through cache misses and a stream mode, in which data is routed directly

between processors without going through local memories. The low-latency, high-bandwidth networks of Raw also allow ILP to be mapped efficiently to Raw. Raw's peak performance can be achieved when data can be operated on without going through local memories in the tiles. For this study, we used standard C to program the kernels. Assembly code was inserted only where necessary to access streaming data through the network. Other programming models, such as decoupled processing, are being developed for Raw and have the potential to improve performance of applications such as those described in this report.

II.F. The PowerPC Architecture

The PowerPC used in this study is the MPC7455 (G4 architecture) operating at 1GHz in an Apple PowerMac system [1][11]. It has four integer units (3 simple + 1 complex), a double-precision floating-point unit, four AltiVec units (simple, complex, floating-point, and permute), a load/store unit, and a branch-processing unit. The MPC7455 processor's 32-bit superscalar core contains a three-issue (plus branch) capability, a 128-bit wide AltiVec unit, a 256-kilobyte on-chip L2 cache, and a 64-bit MPX Bus/60x Bus. The PowerPC G4 provides a vector instruction set extension (AltiVec), which was used manually to achieve the G4 results shown in Section 0. The AltiVec instruction set allows four 32-bit floating-point operations to be specified and executed in a single instruction.

II.G. Performance Models

In this section, simple performance models used to estimate the upper bound of the performance of the architectures are described. We model computation and memory bandwidth. Memory latency is not modeled since these architectures can generally hide memory latency on the kernels used in this study.

Table 1. Architecture peak throughputs (32-bit words per cycle)

	PowerPC	M32R/D	VIRAM	Imagine	Raw
On-chip Memory Read/Write	8 (Cache)	1 (DRAM)	8 (DRAM)	16 (SRF)	16 (Cache)
Off-chip DRAM Read/Write	0.53	0.125	2 (Using DMA)	2	28
Computation per cycle	9	1	8	48	16

Table 1 shows the DRAM memory and ALU throughput for 32-bit data elements that each architecture can support. It should be noted that both memory and ALU throughput are functions of these particular implementations and are not functions of the architectures themselves. However, the architectures provide the means to exploit the throughput supported by the implementation. It should also be noted that memory bandwidth reported is for the nearest DRAM. For V-IRAM, DRAM is on-chip, while the nearest DRAM is off-chip for Imagine and Raw.

III. KERNEL IMPLEMENTATIONS

In this section, the data-intensive kernels used to evaluate the architectures are described. The techniques used to maximize the performance of the kernels on V-IRAM, Imagine, and Raw are presented. The kernels chosen were all identified as representative of processing bottlenecks in Lockheed Martin's Aegis radar system. Lockheed Martin's Maritime Surveillance Systems group in Moorestown, New Jersey, selected the kernels and provided baseline implementations as part of the SLIIC project.

III.A. Corner Turn

The corner turn is a matrix transpose operation that tests memory bandwidth. The data in the source matrix is transposed and stored in the destination matrix. The matrix size used for this report, which was chosen to be larger than Imagine's SRF (128 KB) and Raw's internal memories (2 MB), but smaller than V-IRAM's on-chip memory (13 MB), is 1024 x 1024 with 4-byte elements.

Naive implementations of the corner turn can have poor performance because cache performance can be bad and strided data accesses degrade DRAM bandwidth. In conventional cache-based processor systems, tiling is used to reduce cache misses.

Our VRAM corner turn uses a tiling algorithm with a 16 x 16 element matrix. Tiling allows the vector registers to be used for temporary storage between the loads and stores. We used strided load operations with padding added to the matrix rows to avoid DRAM bank conflicts. Initial load latencies are not hidden. Stores are done sequentially from the vector registers to the memory.

Since the term "column (row)" is used for both matrices and memories, we will distinguish them by denoting them as "matrix column (row)" and "memory column (row)."

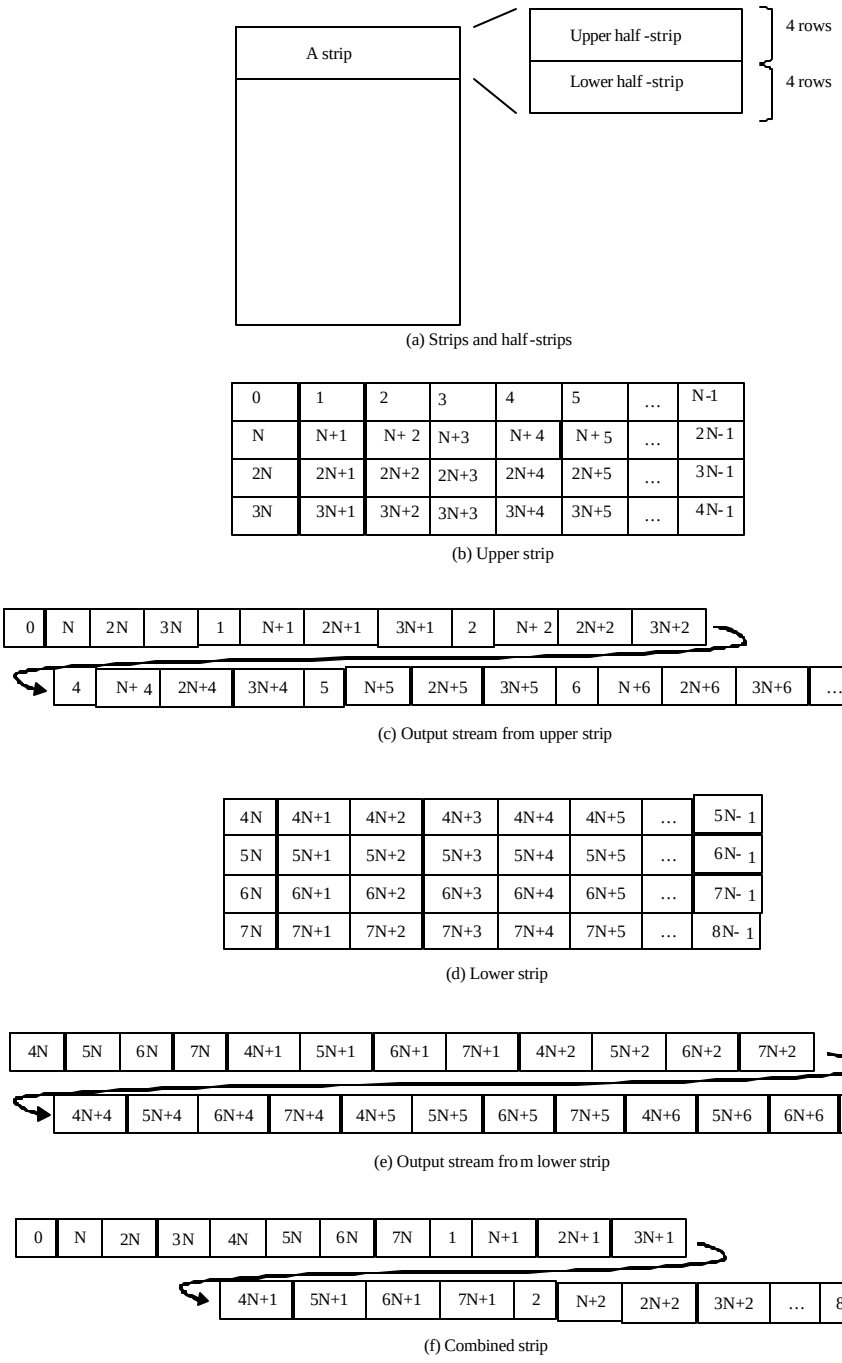


Figure 15. Corner turn on Imagine

The tile size we chose for our corner turn implementation on V-IRAM is a 16 x 16 element matrix. The selection of tile size depends on the number of vector registers and the memory configuration. In the load operations of our implementation, each column in the tile is loaded into each register in strided mode. Then, the data in the registers are stored as a row in sequential mode. Even though we use the strided loads, the effective performance is as good as when sequential accesses are used because of the method described below.

When the first column in the tile is loaded, the load operation for each data stalls a few cycles while the memory column is accessed. However, when the second column in the tile is accessed, if the memory columns accessed previously are not pre-charged, then, the second matrix column can be accessed in one cycle. This is true for the third and all of the remaining matrix columns in the tile. The V-IRAM provides up to eight open columns. Thus, it is possible to keep the columns open when all of the memory columns accessed are in different memory row/wing combinations.

To limit the number of open columns to eight, we first partition the tile into two half-tiles: upper and lower. All data in the upper half-tile is read into the registers before the data in the lower half-tile. Since the size of the upper half-tile is 8 x 16, it is possible to keep all the columns open. Also, we need to ensure that the wing-matrix combination does not appear more than once for the half-tile; otherwise, the previously opened column must be pre-charged and performance is degraded significantly. We used matrix padding to place data in different memory rows and wings. By using this algorithm, the performance of the stride access can be as fast as sequential access.

On the Imagine processor, we use the following technique to leverage the streaming capabilities. We partition the matrix into strips of data. Each strip consists of eight rows of data. For each strip, we read the data in the strip and do a transpose. Since the data in the source matrix is 8 x N elements, where N is the number of columns in the matrix, the transposed data is N x 8. The transposed data is stored in the destination matrix. This is explained in more detail in the following paragraphs.

The strip is conceptually partitioned into two half-strips: an upper half and a lower half (see Figure 15). We first perform the corner turn for the upper half-strip ((b) and (c)). We read the four matrix rows and do the transpose using communication units in the clusters. For this operation, four input streams and one output stream are used. Since the rows are read sequentially, there is no performance degradation. The same operation is performed for the lower half-strip ((d) and (e)). Then, the two output streams are read and permuted using the communication unit in the clusters (f). The strip is written into the destination matrix. During the write operation, the unit of data is eight elements and the stride of data accesses is N. When each row in the strip is written, the data is sequentially stored, thus, we can obtain the maximum possible bandwidth. The cycles lost due to the stride mode for the write operation is inevitable since it is a characteristic of DRAM that the pre-charge time is required whenever memory rows are accessed.

Our corner turn on Raw uses one load and one store operation for each DRAM-to-DRAM transfer. The algorithm, designed at MIT and implemented at USC/ISI, was developed to ensure that all 16 Raw tiles are doing a load or store during as many cycles as possible and to avoid bottlenecks in the static networks and data ports. The algorithm operates on 64x64 word blocks that fit in a single local tile memory. Main memory operations are all done sequentially to maximize memory bandwidth since the transpose can be done in local memories, where all accesses are done in a single cycle.

III.B. Coherent Side-Lobe Canceller (CSLC)

CSLC is a radar signal processing kernel used to cancel jammer signals caused by one or more jammers. Our CSLC implementation consists of FFTs, a weight application (multiplication) stage, and IFFTs (inverse FFTs). Most of the computation time is spent on the FFT and IFFT operations.

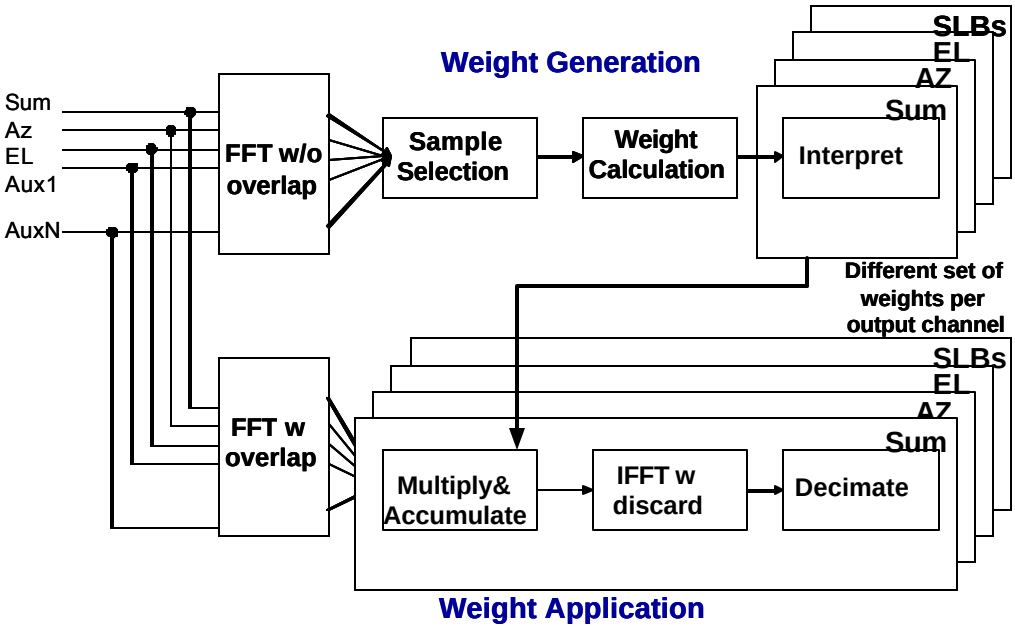


Figure 16. Coherent sidelobe canceller (CSLC)

The block diagram of the signal processing is shown in Figure 16. The operations in the upper half of the figure are known as weight calculations and the operations in the lower half are weight applications. To cancel the side-lobe, the weight factor is calculated using the signal from the auxiliary channel. Then, the main signal is partitioned into several sub-bands in the time domain. Each sub-band is then converted to the frequency domain using the FFT (sub-banding). Weight factors are multiplied with the output of the FFT operation to cancel the side-lobe. An inverse FFT is later performed on the output data. Most of the computation time is spent on the FFT and IFFT operations. In our implementation, only the weight application is implemented.

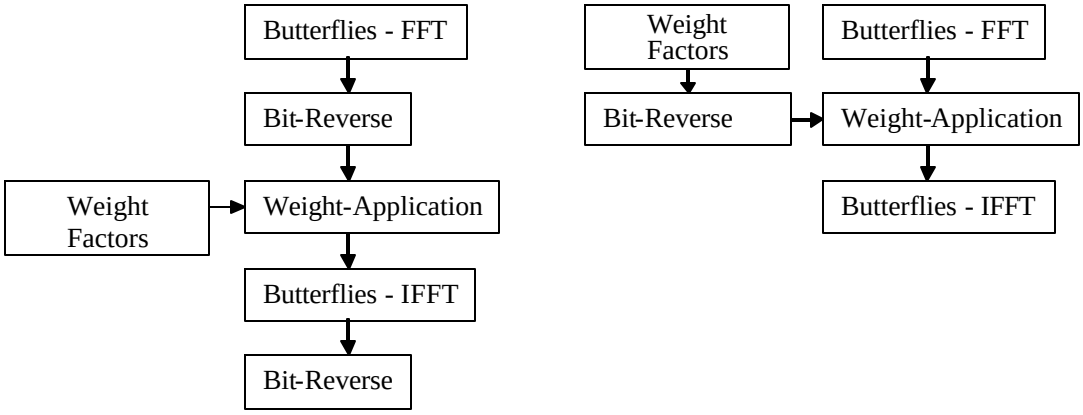
There are four input channels: two main channels and two auxiliary channels. Each channel has 8K samples per processing interval. All computations are done using single-precision floating-point operations. The data is partitioned into 73 overlapping sub-bands, each of which contains 128 samples, so 128-sample FFTs are used.

To improve CSLC performance, we used several techniques: a combination of radix-4 and radix-2 FFT, hand optimization of assembly code for the FFT operation, reducing the number of bit-reverse operations, and eliminating load-store operations between computational stages. Since the majority of computation time on the CSLC is spent on the FFT operation, we improved the performance of the FFT by using the appropriate FFT algorithms for each architecture.

On V-IRAM, a radix-4 FFT is used. Note that since the size of the FFT for the CSLC is 128, which is not power of 4, we used three stages of radix-4 FFT and one stage of radix-2 FFT. Since the current version of the V-IRAM compiler does not vectorize the FFT code written in C optimally, we manually wrote assembly code for the FFT to obtain the maximum performance using vector instructions. For example, there are instructions that are suitable for the FFT butterfly that the current compiler does not use for the FFT compilation, such as *vhalfup*, which shuffles data between two vector registers.

On the Imagine processor, the radix-2 FFT is used. After each butterfly operation, the data is exchanged among clusters using a cluster communicator to arrange data appropriately. In addition to the optimization of the FFT itself, we also removed the bit-reverse operations. Instead of bit-reversing the result of the FFT, the weight factors are bit-reversed in the weight application processing. This is shown in Figure 17.

On the Raw processor, we did not hand-optimize our Raw FFT implementation. A C implementation of the radix-2 FFT is used for Raw because it provided better performance than the radix-4 FFT because of register spilling in the radix-4 FFT. The Raw implementation of CSLC exploits data parallelism by doing independent FFTs on different tiles in parallel.



(a) Conventional (b) Bit-reversed weight factors

Figure 17. New CSLC implementation

III.C. Beam Steering

Beam steering is a radar processing application that directs a phased-array radar in an arbitrary direction without rotating the antenna physically. In a conventional radar system, to send and receive signal from a specific direction, the antenna must be rotated in that direction. This operation needs electrical power to drive a motor and the steering speed is limited by the inertia of the antenna.

Beam steering is used for agile steering of phased array antennas. Figure 18 shows two-dimensional antenna arrays and Figure 19 shows a one-dimensional beam steering operation. In the system, many small antenna elements transmit the signal with different phases. In the figure, each of the three antenna elements transmits a signal with phase shift of d * sin ? between adjacent elements. By choosing phases, the antenna direction can be controlled. The computation of the phase for each antenna element involves many load, store and arithmetic operations.

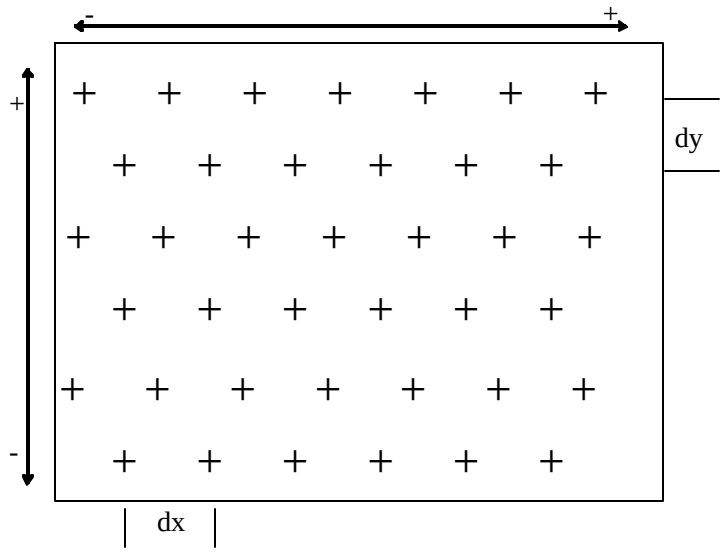


Figure 18. Phase array for beam steering

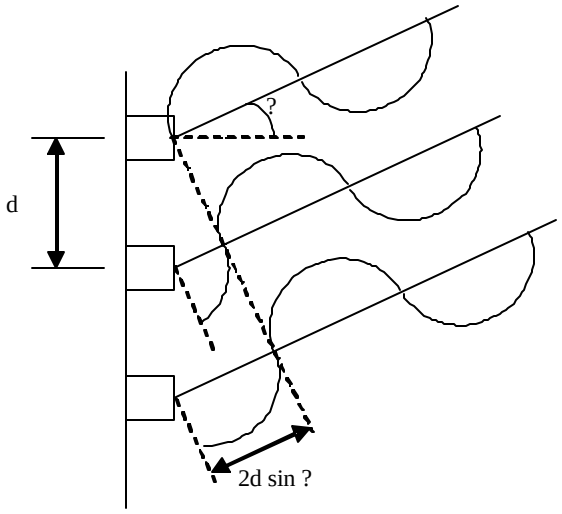


Figure 19. One dimensional view of beam steering operation

In our implementation, the following parameters are used. The number of antenna elements is 1608. Each element can direct the signal up to 4 directions per dwell where a dwell is a period. The phase needs to be calculated for each direction. Depending on the signal frequency and temperature, calibration data needs to be incorporated in the calculation of the phases. In our implementation, four calibration bands are processed.

We did hand-vectorization of the main portion of the beam steering on V-IRAM. Note that the current compiler is still a prototype and it may be able to vectorize these in the future. Since the same processing is performed for each data, the data is fed to the vector unit, which computes output data.

For the Imagine processor, a manually optimized kernel was written to maximize cluster ALU utilization. The input data streams are loaded into the stream register file and supplied to the clusters. The results are written back to memory through the register file.

The beam steering processing on each data is independent. Thus, on Raw, we partition the data among 16 tiles and each tile processes its own data. Input data is streamed through the static network and is operated on directly from the network.

III.D. Digital Target Generator (DTG)

The DTG application generates artificial radar targets used for testing actual radar systems in the field. The original algorithm is shown in Figure 20, where *num_channels* represents the number of radar channels and *num_rows* represents the number of rows of transmit/receive modules processed by a single processing element. In this implementation, *number_of_rows* is set to eight, *num_channels* is set to five, and *num_samples_per_channel* is set to 1,024.

```
Input: wfm // num_channels by num_samples_per_channel array; each element of the
array is a complex number
       Noise // num_channels by num_samples_per_channel by number_of_rows array;
each element of the array is a complex number
Output: out, // three dimensional array; each element of the array is a complex number
1 Initialize phase, phase_inc
2 For c=1 to num_channels
3     for j = 1 to number_of_rows
4         For i=1 to num_samples_per_channel
5             Read wfm(c,i);
6             phase = phase + phase_inc;
7             output(c,i,j).real = sin(phase+wfm(c.i).imaginary) * wfm(c,i).real; // sin()
implementation is shown in Appendix A
8             output(c,i,j).imaginary = cos(phase+wfm(c.i).imaginary) * wfm(c,i).real;
```

Figure 20. Simplified original DTG algorithm

The original, naïve algorithm for DTG performed more memory transfers and trigonometric functions than necessary. To ensure that we performed performance analysis on a well-optimized algorithm, we transformed the original algorithm to that shown in Figure 21. The improved algorithm produces the same results.

```
Input: wfm // num_channels by num_samples_per_channel array; each element of the
array is a complex number
       Noise / /num_channels by num_samples_per_channel by number_of_rows array;
each element of the array is a complex number

Output: out // three-dimensional array; each element of the array is a complex number
1 Initialize phase, phase_inc
2 For c=1 to num_channels
3     for i=2 to number_of_rows
4         initialize ccc(i) and sss(i);
5     for i=1 to num_samples_per_channel
6         Read wfm(c,i); Read noise(c,i,1);
7         phase = phase + phase_inc;
8         ss = sin(phase+wfm(c.i).imaginary) * wfm(c,i).real; // sin() implementation is
shown in Appendix A
9         cc = cos(phase+wfm(c.i).imaginary) * wfm(c,i).real;
10        output(c,i,1).real = cc + noise(c,i, 1).real;
11        output(c,i,1).imaginary = ss + noise(c,i, 1).imaginary;
12            for j = 2 to number_of_rows
13                Read noise(c,i,j);
14                output(c,i,j).real = ccc(j)*cc – sss(j)*ss + noise(c,i,j).real;
15                output(c,i,j).imaginary = ccc(j)*ss + sss(j)*cc + noise(c,i,j).imaginary;
```

Figure 21. Simplified improved DTG algorithm

We implemented the sine and cosine functions using simplified Taylor series with fewer condition checks than general math libraries because of the limited domain of the input arguments. The implemented sine and cosine functions used are shown in Appendix A.

IV. EXPERIMENTAL RESULTS AND ANALYSIS

IV.A. Overview

In this section, the implementation results are presented. Performance of these kernels is obtained by using SLIIC-QL board and cycle-accurate simulators provided by the V-IRAM, Imagine, and Raw teams. For comparison purposes, actual measurements of baseline performance were taken using a single node of a 1 GHz PowerPC G4-based system (Apple PowerMac G4) [1]. AltiVec technology was used for the baseline performance. The Apple cc compiler was used with timing done using the MacOS X system call mach_absolute_time(). We manually inserted AltiVec vector instructions.

Table 2 summarizes key parameters of each processor. Note that the PowerPC and M32R/D are a highly optimized chips implemented with custom logic while the other processors are research chips implemented using standard cells and very small design teams. Thus, if the same level of design effort were applied to these research architectures, we would expect much higher clock rates and density to be achieved.

In Table 3, a summary of the implementation results is shown. The corner turn result for the M32R/D is extrapolated from a 512x512 element matrix, which is the largest corner turn result that can be run on the M32R/D. Figure 22 shows the speedup in terms of cycles for corner turn, CSLC, and beam steering and Figure 24 shows the speedup for DTG. Figure 23 shows the speedup in terms of execution time for corner turn, CSLC, and beam steering and Figure 25 shows the speedup for DTG in terms of execution time. Note that Figure 22 and Figure 23 both use a log scale on the vertical axis.

Table 2. Processor parameters

	PPC G4	M32R/D	VIRAM	Imagine	Raw
Clock (MHz)	1000	80	200	300	300
# of ALUs	8	1	16	48	16
Peak GFLOPS	9	-	3.2	14.4	4.64

Table 3. Experimental results (cycles in 10^3)

	Corner Turn	CSLC	Beam Steering	DTG
PowerPC - AltiVec	30,117	3,204	870	764
M32R/D	5,712	18,735	1,944	-
VIRAM	554	424	35	-
Imagine – slow memory	-	-	-	199
Imagine – Fast memory	1,207	145	87	92
Raw	145	357	19	-

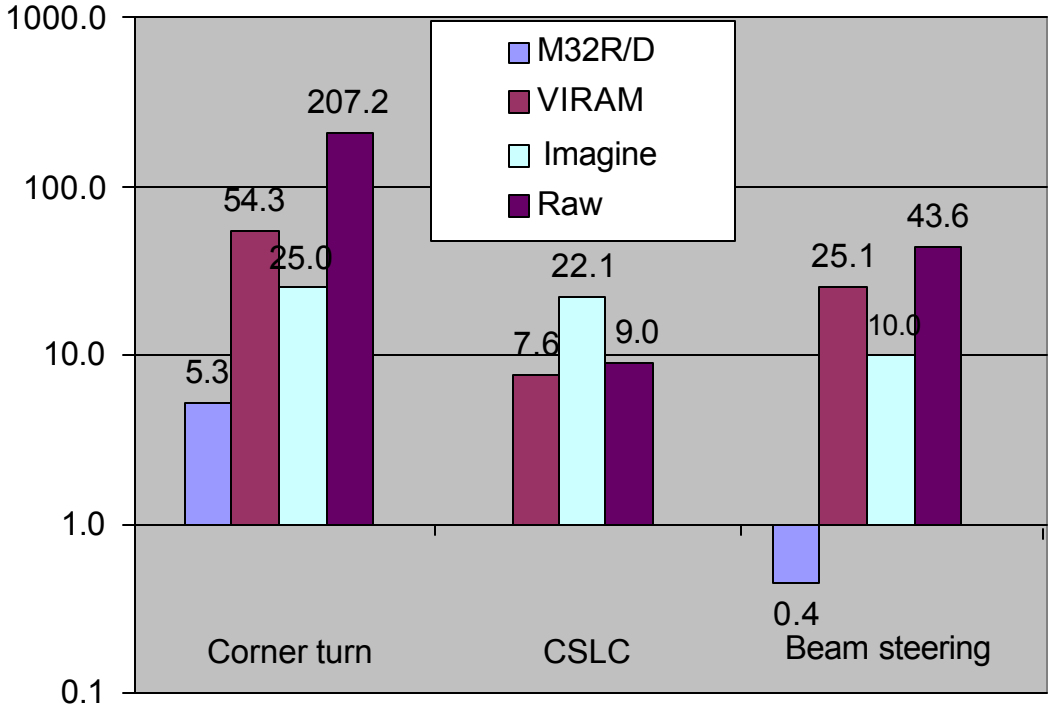


Figure 22. Speedup compared with PowerPC with AltiVec (cycles)

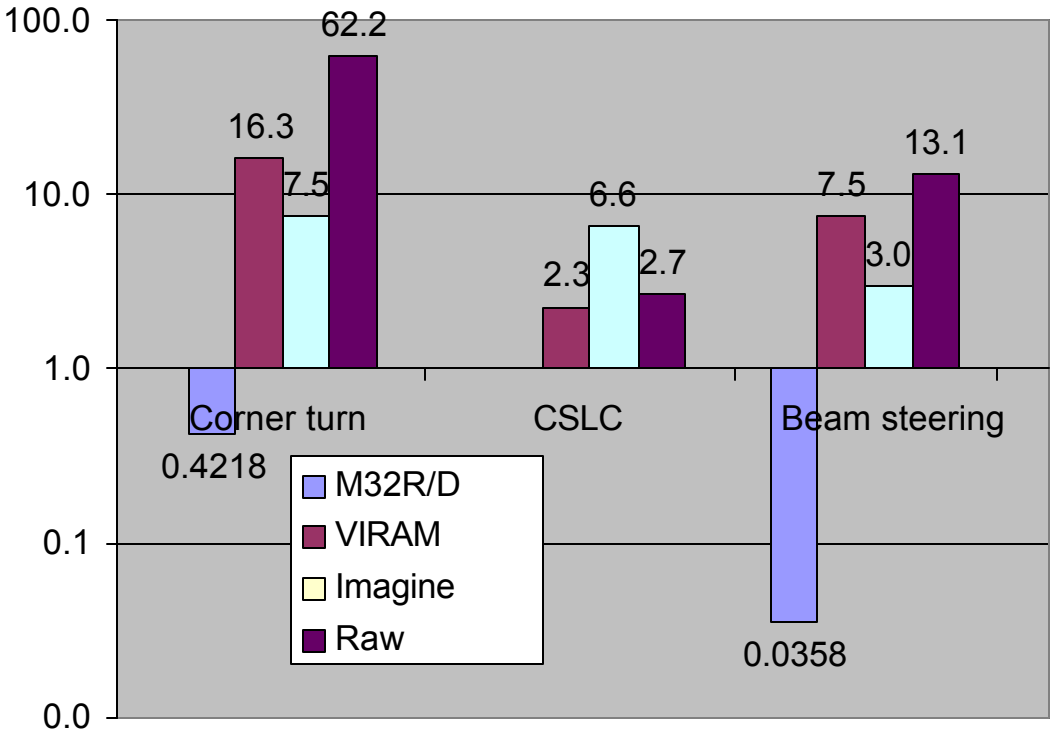


Figure 23. Speedup compared with PowerPC with AltiVec (execution times when PowerPC=1 GHz, M32R/D=80MHz, V-IRAM=200 MHz, and Raw=300 MHz)

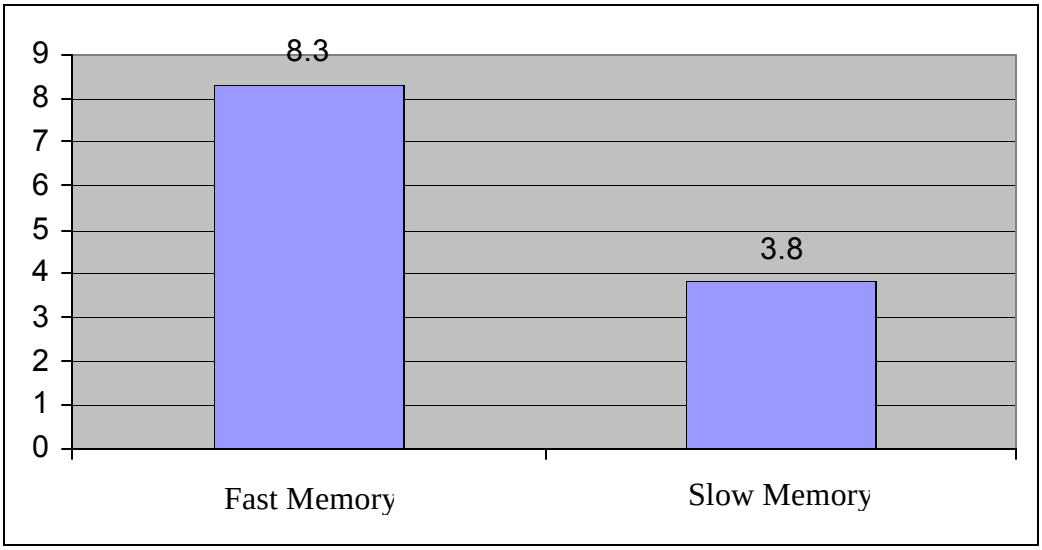


Figure 24. DTG speedup on Imagine compared with PowerPC with AltiVec (cycles)

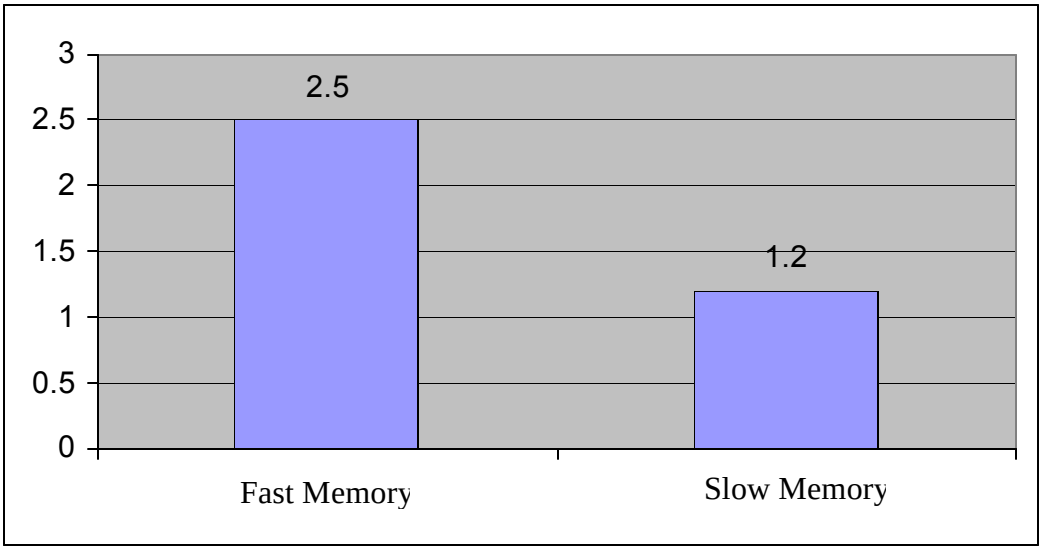


Figure 25. DTG speedup compared with PowerPC with AltiVec (execution times when PowerPC=1 GHz and Imagine=300 MHz))

IV.B. Corner Turn Performance

All three architectures except the M32R/D provided speedups of more than 20 compared with a PowerPC system in terms of number of cycles. Corner turn performance is mostly a measure of memory bandwidth, which is not a direct property of an architecture, but rather a

function of the number of pins in the package. However, the corner turn does demonstrate an architecture's ability to leverage memory bandwidth that does exist. Since V-IRAM has on-chip DRAM, the kernel measures on-chip bandwidth. On the Imagine and Raw architectures, w off-chip memory is stressed.

The performance of the corner turn on the M32R/D is about 37% of what would have been expected from peak cache bandwidth. Since the peak memory-cache bandwidth is higher than cache-CPU bandwidth, the memory-cache bandwidth is not a bottleneck. Instead, the cache-CPU bandwidth is bottleneck for M32R/D. We believe one of the main reasons for the difference between the peak performance and obtained result is the added cycles for memory access latency.

The performance of the corner turn on V-IRAM is about half of what would have been expected from peak memory bandwidth. About 21% of the total cycles are overhead due to DRAM pre-charge cycles (which would be mostly hidden with sequential accesses) and TLB (Translation Look-aside Buffer) misses, and 24% are due to a limitation in strided load performance imposed by the number of address generators.

On Imagine, we assume the memory clock is the same frequency as the processor clock. Imagine has two address generators that provide two words per clock cycle. Note that the number of address generators is a processor implementation choice and is not a limitation of a stream architecture. Since the goal of the Imagine project was to demonstrate how memory traffic could be reduced, the Imagine team chose not to implement a high-bandwidth memory interface.

If the network ports were used to transfer data between SRF and an external memory for the corner turn, the performance would be the same since the network port has a peak performance of two words per cycle.

87% of the cycles in the Imagine corner turn are due to memory transfers that are close to the maximum theoretical performance. The remaining 13% of the execution cycles are due to non-overlapped cluster instructions. Conceptually, the kernel instructions should be fully overlapped with memory accesses, but a limitation induced by the stream descriptor registers prevented full software pipelining in our implementation.

The Raw chip implementation actually provides enough main memory bandwidth that it is not the performance limiter for our corner turn implementation. Load/store issue rates and local memory bandwidth limit performance. 16 instructions per cycle are executed on the Raw tiles, and the static network and DRAM ports are not a bottleneck. The performance we achieved is nearly identical to the maximum performance predicted by the instruction issue rate. Memory latency is fully hidden (except for negligible start-up costs).

IV.C. CSLC Performance

CSLC mainly consists of FFTs and matrix-vector multiplication. Since the FFT length is 128, the working set fits into local memory, and the performance of the CSLC depends primarily on ALU performance for Imagine and Raw.

Since the M32R/D does not support hardware floating-point operations, the CSLC on the M32R/D is an integer version of the CSLC. Therefore, the performance on M32R/D is not compared with other implementations directly. The obtained performance is about 13% of the peak performance of the M32R/D. We believe one of the reasons for this low utilization is that many overhead instructions are needed in addition to the computation instructions such as maintaining loop variables, address calculations, and temporary variables.

Our V-IRAM CSLC implementation result shows that it takes about 3.6 times longer than what is predicted by peak performance. The first factor reducing performance is overhead instructions. Instructions are needed to perform the FFT shuffles and increase the number of cycles by a factor of 1.67. The second factor that reduces FFT performance is ALU utilization. Since the second vector arithmetic unit in V-IRAM cannot execute vector floating point instructions, performance on the FFT is reduced by a factor of 1.52. Finally, memory latency and vector startup costs increase performance by a factor of 1.41.

The CSLC on Imagine and Raw uses radix-2 FFTs. On Raw, radix-2 was used to avoid register spilling encountered in the radix-4 FFT. On Imagine, the radix-4 FFT provides better performance (about 34%), but a complete CSLC was not implemented within the scope of SLIIC. The number of operations (including loads and stores) in the radix-2 FFT is about 1.5 times of the number in the radix-4 FFT. So care should be given when the performances of the CSLC on Raw and Imagine are compared with CSLC performance on other architectures.

Imagine has the best performance of the three architectures on CSLC. This is because it is a computation-intensive kernel for which the working sets fit in the stream register files. Although the data access patterns for FFT are challenging for any architecture, the streaming execution model of Imagine is able to reduce memory operations and Imagine functions as intended on this kernel. Overall, performance achieved on CSLC on Imagine is about 39% of what is predicted by peak performance. While this is much lower than those achieved for many media benchmark kernels, it still allows Imagine to perform about 16 useful operations per cycle; much better than can be achieved on today's superscalar architectures. Performance is reduced by 35% because inter-cluster communication is used to perform parallel FFTs. An alternative implementation, which was not completed under SLIIC, would execute independent FFTs in parallel to eliminate inter-cluster communication overhead.

For the FFT kernel, ALU utilization (as measured by minimum FFT computations / total ALU cycles available) is 34.0%. If we exclude the divider, which is not useful for the FFT, then the utilization is 40.8%. Note that the utilization is on the lower side of the more than 40% obtained in other processing intensive applications [8]. The reason for the relatively low utilization is that the small size of the FFT reduces the amount of software pipelining and start-up overheads are not as well amortized with a small kernel.

On Raw, we implemented a data parallel version of CSLC. The local memory on Raw successfully caches the working sets, and less than 10% of the execution time is spent on memory stalls. Note that most of this stalling could have been eliminated by implementing a streaming direct memory access transfer to the local memory that is overlapped with the computation.

One problem with our data parallel implementation of CSLC on Raw was load balancing. The CSLC is easily parallelized for 16 tiles, however, since the number of data sets is 73, which is not a multiple of the number of tiles, some tiles processed five sets while others processed four sets. About 8% of CPU cycles are idle due to load balancing. The number of sets in a real environment is not fixed at 73. In a real implementation, the input data sets would arrive continuously. Therefore, it is reasonable to assume that Raw could have perfect load balancing in a real implementation. Thus, we report the performance numbers for CSLC on Raw based on an extrapolation that assumes perfect load balancing.

Raw achieves about 31.4% of the peak performance on CSLC. About 26% of the cycles on Raw are consumed by load and store overhead instructions. Cache stalls take 7.4% of the total

cycles. The remaining cycles are consumed by address and index calculations and loop overhead instructions.

If the FFT were implemented using the stream interface that uses static network, cache miss stalls would be hidden, and load and store operations would not be needed. A primitive implementation result suggests an improvement of about 70%.

IV.D. Beam Steering Performance

Beam steering has a small number of computations (5 additions and 1 shift) per output data and a relatively high number of memory accesses (2 loads and one store). On the M32R/D, the lower bound of the computation time is 16% of the total execution time. Note that the only optimization done for the implementation on M32R/D is using optimization flag for the compiler. On other architectures, in addition to the use of the optimization flag, intensive optimization efforts have been performed. We did not invest the same effort in optimizing for the M32R/D after it became clear that it does not provide performance competitive with the other chips for computation-intensive applications. We believe the primary reason for the difference between the lower bound and total execution cycles is overhead instructions used for maintaining loop variables, address calculations, and temporary variables, which could be reduced if more human optimization effort was performed.

On V-IRAM, the lower bound of the computation time is 56% of the simulation time. The difference between the expected and simulation results comes from waiting for the results from previous vector operations and the cycles needed to initialize the vector operations.

On Imagine, the computations and memory accesses for beam steering are overlapped. Performance is limited by memory bandwidth due to the relatively low number of computations per memory access. The load and store operations take 89% of the simulation time. The remaining 11% of execution time is due to the software pipeline prologue.

In an actual signal processing pipeline the beam steering kernel would stream its inputs from the proceeding kernel in the application (e.g., a poly-phase filter bank) and stream its outputs to the following kernel (e.g., per-beam equalization). In such a pipeline the performance of beam steering will not be limited by memory bandwidth, as in the case of this isolated kernel, but rather will be limited by arithmetic performance. On such a streaming application Imagine is expected to achieve a high fraction of its peak performance. If table values were read from the stream register file rather than memory on our kernel, performance would be increased by a factor of about two. The performance of a beam steering algorithm with more computation per data (which is a realistic assumption) could be much higher.

On Raw, we used the static network to stream data from memory while hiding memory latency. In this implementation, loads and stores are not necessary and ALU utilization is very high. It attains 96.6% of the peak performance. The Raw beam steering implementation has the best performance of the three architectures because of the combination of memory bandwidth and high ALU utilization.

IV.E. DTG Performance

We implemented DTG on Imagine and the PowerPC. Table 3 shows the number of cycles and execution time in microseconds. Imagine was simulated with two assumptions: i) the slow-memory case, where the memory bus frequency is one fourth the frequency of the Imagine processor, and ii) the fast-memory case, where the memory bus frequency is the same as the Imagine processor frequency.

In the slow-memory case, DTG takes 198,734 cycles. The lower bound for this case is 174,080 cycles, which is 87.6% of the execution time. In the fast-memory case, where the memory clock speed is the same as the Imagine clock, it takes 92,392 cycles. The lower bound of the DTG on Imagine is 87,040, which is 94% of the simulation time. The rest of the time is due to the gap between memory accesses. Thus, the memory system utilization is 87.6% and 94% for the slow-memory case and the fast-memory case, respectively. The ALU utilization is 4.6% and 9.8% for the slow-memory and fast-memory cases, respectively, which is very low compared with other computation-intensive applications. The reason is that DTG is more memory intensive, and the data supply to the ALU units is far lower than the ALU computation capability.

The PowerPC takes 763,930 cycles. The lower bound of DTG on the PowerPC is 327,218 cycles, which is 42.8% of the measured execution time. The total execution time consists of the following blocks: i) initialization, ii) computation of the first data of eight data elements (Line 6–11), and iii) computation of the remaining seven data elements (Line 14–16).

Let us first analyze the case when the data is in cache. For the initialization, the number of cycles was 5.6% of the total execution cycles. The second part takes 46.6% of the total execution time. The third block (Line 14–16) repeats 1,280 times. This block cannot fit in the first-level cache since the noise is large (81,920 words), so the noise data is in the second-level cache. The number of cycles to bring a cache line (2 words) to the ALU is 9 cycles [11]. The number of load/store instructions needed in each iteration is 28. Each load/store instruction handles four words simultaneously. Since the cache line size is eight words, a cache miss occurs every two instructions. Therefore, there are 14 cache misses in each iteration. Thus, the number of cycles needed for this block is 161,280 cycles. The total number of cycles measured for this part is 165,594 cycles (21.7% of total execution cycles). In addition to this, transfers between memory and cache are needed. This accounts for 26.0% of the total execution time. To find the optimal parameters for prefetching, many combinations of the parameter values were tried and the best setting was chosen.

The speedup of Imagine compared with the PowerPC is 3.8 for the slow-memory case and 8.3 for the fast-memory case in terms of number of cycles. One of the differences comes from the different memory clock speeds and difference data bus widths. The memory clock speed of PowerPC is 266 MHz and the data bus width is two words. Since the bus bandwidth of the Imagine is four words, when the memory clock speed of the Imagine is 250 MHz, the performance ratio is 1.88 (250 MHz * 4 words/(266 MHz * 2 words)). However, when the Imagine memory clock speed is the same as Imagine chip, then, the number of address generators becomes the bottleneck, and only two words per cycle can be transferred. So the performance difference is 3.76 (=1000 MHz * 2 words/(266 MHz *2 words)). This contributes 3.76 (=times of the difference for the fast-memory case.

The other latency is between cache and memory. On Imagine, the memory latency is hidden because memory accesses run ahead of computations and streaming reduces the number of off-ship memory accesses. On the PowerPC, some of the memory access latency is hidden by prefetching, but prefetching cannot hide latency completely for data intensive applications that consume all available memory bandwidth. Main memory latency leads to a performance difference of 1.36.

IV.F. Architecture Comparison

V-IRAM's primary advantage comes from the high bandwidth between the vector units and DRAM without paying the cost (in terms of pins and power) that are required to achieve high bandwidth between chips. V-IRAM is especially suitable for vectorizable applications that can utilize the high bandwidth interface and that are small enough to fit in the on-chip memory. V-IRAM outperformed the G4 AltiVec by more than a factor of 7 on all four of our kernels and showed especially good performance on the kernels that emphasize memory bandwidth. For embedded applications with reasonably sized data sets, the V-IRAM can be used as a one-chip system. If the application size is larger than the on-chip DRAM, the data needs to come from off-chip memory and V-IRAM would lose much of its advantage.

Imagine's high peak performance can be utilized in streaming applications where main memory accesses can be avoided or minimized. The CSLC kernel demonstrates that even when the Imagine ALUs are not fully utilized, performance can be quite high, especially when compared to a commercial microprocessor like the G4 AltiVec. Imagine's stream-based architecture is designed for scalability and power efficiency, and the Imagine architecture has the highest peak performance of the architectures in this study.

Raw also performs best on streaming applications since load and store operations can be eliminated and the static networks provide tremendous on-chip bandwidth. The kernels used in this study do not fully exploit this mode of execution. But we have shown that the tile structure of Raw can be used to utilize the memory bandwidth available from the external ports of Raw. The tile structure also provides flexible support for MIMD and ILP applications.

V. CONCLUSION

In this report, the implementation results of four signal processing kernels (the corner turn, CSLC, beam steering, and digital taget generator) on systems based on one commercial off-the-shelf processor (M32R/D) and two recent research processors (V-IRAM and Imagine) are presented and compared to a commercial PowerPC. The results show that these data-intensive-system architectures have strengths and provide significant performance potential compared to the current generation of superscalar processors with vector extensions.

The performance of some of the kernels on these architectures is limited by DRAM interface bandwidth. In these cases, the computational potential of the processors is not fully exploited. Instead, these data-intensive kernels stress-test the DRAM interface of the processors. This DRAM interface is a design choice influenced by cost rather than a function of these innovative architectures.

The stream programming model, as expressed by StreamC and KernelC, used for Imagine fits some, but not all, of our data-intensive kernels well. The applications best suited for the stream programming model have long sequences of data and data that goes through many stages of processing before the data is stored back to memory. An example of this kind of application is the CSLC. Since the tools for the stream programming are in a research stage, significant human optimization was required.

These emerging architectures did demonstrate that they can be programmed reasonably using high-level languages and existing compilers to obtain adequate performance, while with hand optimization or future compilers, they can achieve performance that far outstrips existing architectures. Furthermore, all of these architectures will scale as technology shrinks far better than today's superscalar processors.

VI. ACKNOWLEDGEMENTS

The authors gratefully acknowledge the extraordinary support of the UC Berkeley IRAM team, the Stanford Imagine team, and the MIT Raw team for the use of their compilers, simulators, and computational kernels and their generous help. This study obviously would not have been possible without their generous support.

VII. REFERENCES

[1] Apple, http://www.apple.com/powermac/, 2002.

[2] S. Chatterjee and S. Sen, "Cache-efficient matrix transposition," *Sixth International Symposium on High-Performance Computer Architecture*, Touluse, France, 2000.

[3] Gordon, M., Thies,W, Karczmarek, M., Lin, J., Meli, A. S., Lamb, A. Leger, A. C., Wong, J., Hoffmann, H., Maze, D., Amarasinghe, S. "A Stream Compiler for Communication-Exposed Architectures," *MIT Tech. Memo TM-627*, Cambridge, MA, March 2002.

[4] Gupta, A., Hennessy, J. L., Gharachorloo, K. T., Mowry, Weber, W. D. "Computative Evaluation of Latency Reducing and Tolerating Techniques," *Proc. 18th Annual International Symposium on Computer Architecture*, Toronto, May 1991.

[5] Hennessy J., Patterson, D. A., *Computer Architecture: A Quantitative Approach*, 2nd Edition, Morgan Kaufmann Publishers, Inc., 1996.

[6] Khailany, B., Dally, W. J., Rixner, S., Kapasi, U. J., Mattson, P., Namkoong, J., Owens, J. D., Towles, B., Chang A. "Imagine: Media Processing with Streams," *IEEE Micro*, March/April, pp 35-46, 2001.

[7] Kozyrakis, C. "Scalable Vector Media-processors for Embedded Systems," Ph. D. dissertation, UC Berkeley, May 2002.

[8] Kapasi, U., Dally, W. J., Rixner, S.,. Owens, J. D., Khailany, B. "The Imagine Stream Processor," *International Conference on Computer Design*, Freiburg, Germany, September 2002.

[9] Kozyrakis, C., Patterson, D., "Vector Vs. Superscalar and VLIW Architectures for Embedded Multimedia Benchmarks," 35th *International Symposium on Microarchitecture*, Istanbul, Turkey, November 2002.

[10] *Mitsubishi Microcomputers, M32000D4BFP-80 Data Book*, http://www.mitsubishichips.com/data/datasheets /mcus/ mcupdf/ds/e32r80.pdf.

[11] Motorola, *MPC7450 RISC Microprocessor Family User's Manual*, February 2003.

[12] Owens, J. D., Rixner, S., Kapasi, U. J., Mattson, P., Towles, B., Serebrin, B., Dally, W. J., "Media Processing Applications on the Imagine," *Stream Processor Proceedings of International Conference on Computer Design*, Freiburg, Germany, September 2002.

[13] S. A. Przybylski, Cache and Memory Hierarchy Design: A Performance-Directed Approach, Morgan Kaufmann Publishers, San Mateo, CA, 1990.

[14] Rixner, S., Dally, W. J., Kapasi, U. J., Khailany, B., Lopez-Lagunas, A., Mattson, P. R., Owens, J. D. "A Bandwidth-Efficient Architecture for Media Processing," *31st Annual International Symposium on Microarchitecture*, Dallas, Texas, November 1998.

[15] Smith, A. J. "Cache Memories," *Computing Surveys*, Vol. 14, No. 3, pp 473-530, 1982.

[16] Suh, J., Crago, S.P. "PIM-based and Stream Processor-based Processing for Radar Signal Applications," *MSP 02*, Austin, TX, 2002.

[17] Suh, J., Crago, S. P., Li, C., Parker, R. "A PIM-based Multiprocessor System," *International Parallel and Distributed Processing Symposium*, San Francisco, CA, 2000.

[18] Suh, J., Kim, E.-G., Crago, S. P. "A Performance Comparison of PIM, Stream Processing, and Tiled Processing on Signal Processing Applications," *ISCA03*, San Diego, CA, June 2003.

[19] Taylor, M. B., Kim, J., Miller, J., Wentzlaff, D., Ghodrat, F., Greenwald, B., Hoffmann, H., Johnson, P., Lee, W., Saraf, A., Shnidman, N., Strumpen, V., Amarasinghe, S., Agarwal, A. "A 16-issue multiple-program-counter microprocessor with point-to-point scalar operand network," *Proceedings of the IEEE International Solid-State Circuits Conference*, February 2003.

[20] Taylor, M. B., Lee, W., Amarasinghe, S., Agarwal, A. "Scalar Operand Networks: On-chip Interconnect for ILP in Partitioned Architectures," *International Symposium on High Performance Computer Architecture*, February 2003.

APPENDICES

Appendix A. Sin Computation Algorithm Used in the DTG Implementation

```
#define C1 0.0083333337680
#define C2 -0.00019841270114
#define C3 0.0000027557314297
#define C4 -0.000000025050759689
#define C5 0.00000000015896910177
#define C6 -0.16666667163

#define S1 0x7fffffff
#define S2 0x32000000

sin(sin, x)
// Input: x
// Output: sin
        ix = asint(x);
        ix = ix & S1;
        z0 = (ix<S2);
        z1 = (ftoi(x)==0);
        z1 = (z0 & z1 & 0x1);
        z0 = 1 - z1;
        z =  x*x;
        v =  z*x;
        r = C1 +z*(C2+ z*(C3+z*(C4+z*C5)));
        sin = x + v*(C6+z*r) * itof(z0);
```

Appendix B. Acronyms

ALU	Arithmetic Logic Unit
API	Application Programming Interface
CPU	Central Processing Unit
CSLC	Coherent Side-Lobe Canceller
DARPA	Defense Advanced Research Projects Agency
DRAM	Dynamic Random Access Memory
DSP	Digital Signal Processor
DTG	Digital Target Generator
EMBC	Embedded Microprocessor Benchmark Consortium
EEPROM	Electronically Erasable Programmable Read Only Memory
FFT	Fast Fourier Transform
FPGA	Field Programmable Gate Array
GFLOPS	Giga-FLoating Operations Per Second
GOPS	Giga-Operations Per Second
IFFT	Inverse Fast Fourier Transform
ILP	Instruction Level Parallelism
MB	Megabytes
MIMD	Multiple Instruction, Multiple Data
MIPS	Million Instructions Per Second
PC	Personal Computer
PCI	Peripheral Component Interface
PIM	Processor In Memory
PPC	PowerPC
RISC	Reduced Instruction Set Computer
QL	Quick Look
SDRAM	Synchronous Dynamic Random Access Memory
SIMD	Single Instruction, Multiple Data
SLIIC	System-Level Intelligent Intensive Computing
SRAM	Synchronous Random Access Memory
SRF	Stream Register File
TLB	Translation Look-aside Buffer
V-IRAM	Vector Intelligent Random Access Memory

Appendix C. Publications

The following publications were supported or partially supported by the SLIIC project.

J. Suh, D. Kang, and S. P. Crago, "A Communication Scheduling Algorithm for Multiple-FPGA Systems," IEEE Symposium on Field Programmable Custom Computing Machines (FCCM) 2000, Napa, CA, April 2000.

J. Suh and V. K. Prasanna, "An Efficient Algorithm for Large-Scale Matrix Transposition," International Conference on Parallel Processing (ICPP), Toronto, Canada, August 2000.

J. Suh, S. P. Crago, C. Li, and R. Parker, "Distributed Corner Turn on a PIM-Based Multiprocessor," Fourth Annual Workshop on High Performance Embedded Computing (HPEC), Cambridge, MA, September 2000.

J. Suh, M. Zhu, C. Li, S. P. Crago, S. F. Shank, R. H. Chau, W. J. Mazur, and R. Pancoast, "Implementations of Real-time Data Intensive Applications on PIM-based Multiprocessor Systems," Joint Workshop on Parallel and Distributed Real-Time Systems and Embedded High Performance Computing, San Francisco, CA, April 2001.

J. Suh, C. Li, S. P. Crago, and R. Parker, "A PIM-Based Multiprocessor System," International Parallel and Distributed Processing Symposium, San Francisco, CA, April 2001.

J Suh, D. Kang, and S. P. Crago, "Efficient Algorithms for Fixed-Point Arithmetic Operations In An Embedded PIM," World Multi-Conference on Systematics, Cybernetics, and Informatics, Orlando, FL, July 2001.

J. Suh, S. P. Crago, C. Li, and R. Parker, "PIM- and Stream Processor-Based Systems," Fifth Annual High Performance Embedded Computing Workshop, Cambridge, MA, November 2001.

J. Suh and S. P. Crago, "PIM- and Stream Processor-based Processing for Radar Signal Applications," The Third Workshop on Media and Streaming Processors in conjunction with The 34th International Symposium on Microarchitecture, Austin, TX, December 2001.

J. Suh and V. K. Prasanna, "An Efficient Algorithm for Out-of-Core Matrix Transposition," IEEE Transactions on Computer, April 2002.

J. Suh, E.-G. Kim, S. P. Crago, L. Srinivasan, and M. C. French, "A Performance Analysis of PIM, Stream Processing, and Tiled Processing on Memory-Intensive Signal Processing Kernels," International Symposium on Computer Architecture, San Diego, CA, June 2003.

A Communication Scheduling Algorithm For Multi-FPGA Systems*

Jinwoo Suh, Dong-In Kang, and Stephen P. Crago

University of Southern California Information Sciences Institute
4350 N. Fairfax Drive, Suite 770, Arlington, VA 22203
{jsuh, dkang, crago}@isi.edu

Abstract

For multiple FPGA systems, the limited number of I/O pins causes many problems. To solve these problems, efficient communication scheduling among FPGAs is crucial for obtaining high CLB utilization. In this paper, we provide a heuristic for the NP-complete scheduling algorithm. The experimental results show that our algorithm generates excellent communication schedules: more than 90% of the randomly generated problem instances were scheduled with less than 20% overhead compared with an optimal algorithm. The execution time of the scheduling algorithm is two orders of magnitude less than the optimal scheduling algorithm.

1. Introduction

In this paper, we discuss communication scheduling on multiple FPGA systems. Since the number of CLBs is proportional to the chip area ($O(l^2)$) while the number of I/O pads is proportional to the chip perimeter ($O(l)$), where l is the length of a chip, the relative I/O bandwidth is getting smaller as the number of CLBs has been increasing continuously. Underutilization of I/O bandwidth exasperates the problem. To support a given I/O rate, pin bandwidth underutilization increases the number of FPGAs needed. This increase in the number of FPGAs causes the parts cost to increase and longer development time[1]. To address these problems, we propose an efficient communication algorithm that uses the available I/O pins efficiently to increase CLB utilization.

2. Problem Definition

An application is partitioned into N tasks. Each task, τ_v, has a computation time, $T(\tau_v)$, $0 \le v \le N-1$. Each task is mapped to an FPGA, F_i, $0 \le i \le F-1$, where F is the number of FGPAs. If a task, τ_v, on F_j needs data from a task, τ_u, $0 \le u \le N-1$, on F_i $(i \ne j)$, then communication must be performed between F_i and F_j before τ_v can be performed. The communication takes $T(e_{u,v})$, where $e_{u,v}$ is the required communication between τ_u and τ_v. The communication is performed through the I/O pins of the FPGAs. Let us denote a set of I/O pins that is used for the communication as a *channel*, C_x, $0 \le x \le C-1$, where C is the number of channels. A channel cannot be used by two communications simultaneously and is non-preemptive. For simplicity, we assume that the channels are unidirectional. The object is to find a communication schedule that has minimum latency for given tasks and channels.

Theorem 2-1: *The scheduling of communication for a multiple FPGA system is NP-complete.*

The theorem can be proved by reducing it to flowshop scheduling[2]. The proof is omitted due to space limitations.

3. Our Algorithm

In this algorithm, a heuristic weight value $w(\tau_i)$, $w(e_{i,j})$ of a task τ_i, and an edge $e_{i,j}$ are used to determine the priority of the communication edge in scheduling. When multiple communication tasks compete for a communication channel at a given time, the one having the largest weight value is chosen for scheduling. The algorithm is shown in Figure 1. The algorithm consists of two steps: (i) evaluation of weights of tasks in breadth-first search fashion (*Calculate_Weight*) and (ii) scheduling of communication edges in each channel in bottom-up fashion (*Largest_Weight_First*).

Algorithm Calculate_Weight(τ_i)
(1) If *visited*(τ_i) = True
(2) Return;

* Effort sponsored by Defense Advanced Research Projects Agency (DARPA) through the Air Force Research Laboratory, USAF, under agreement numbers F30602-99-1-0521, F30602-97-1-0222, and F33615-98-C-1320. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not interpreted as necessarily representing the official policies or endorsement, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), Air Force Research Laboratory, or the U.S. Government.

(3) $w(\tau_i) = 0$;
(4) For each outgoing edge of τ_i, $e_{i,j}$
(5) if *visited*(τ_j) = False
(6) *Calculate_Weight*(τ_j);
(7) $w(e_{i,j}) = w(\tau_j) + T(e_{i,j})$;
(8) $w(\tau_i) = w(\tau_i) + T(e_{i,j})$;
(9) $w(\tau_i) = w(\tau_i) + T(\tau_i)$;
(10) *visited*(τ_i) = True;
(11) For each incoming edge of τ_i, $e_{k,i}$
(12) *Calculate_Weight*(τ_j);

Algorithm Largest_Weight_First(*DAG*)
(1) For all edge *e* in the DAG
(2) *r(e)* = -1;
(3) For each channel C_x in the system
(4) *Schedule_Channel*(C_x);

Algorithm Schedule_Channel(C_x)
(1) If scheduled(C_x) = Yes
(2) Return;
(3) For each edge e_{kl} in C_x
(4) *Get_Ready_Time*(e_{kl});
(5) ***S*** = set of edges in C_x;
(6) *t* = smallest ready time of edges in ***S***;
(7) While (***S*** is not empty)
(8) ***S***' = set of schedulable edges at *t* in ***S***;
(9) If (***S***' is not empty)
(10) Schedule an edge *e* in ***S***' with largest weight;
(11) ***S*** = ***S*** – {*e*}; ***S***' = ***S***' - {*e*};
(12) *t* = *t* + *T*(*e*);
(13) else
(14) *t* = smallest ready time of edges in ***S***;
(15) scheduled(C_{xy}) = Yes;

Algorithm Get_Ready_Time(e_{kl})
(1) If $r(e_{kl}) \geq 0$
(2) Return;
(3) If all incoming edges to τ_k are not scheduled
(4) For each channel containing an incoming edge to τ_k
(5) *Schedule_Channel*(C_{xy});
(6) $r(e_{kl}) = T(\tau_k)$ + largest finish time of incoming edges to τ_k;

Figure 1. Communication schedule generation algorithm

The total complexity of our heuristic algorithm is O(*M* + *N* log *N*), where *M* is the number of tasks in the graph, and *N* is the number of edges.

4. Experimental Results and Conclusions

As a validation of our heuristic approach, we generated 100 random trees for each of 4 different tree sizes and ran experiments and compared it with optimal solutions on a SUN workstation ULTRA 30. Also, the execution times of the two algorithms were measured. These are shown in Figure 2 and 3. The experimental results show that our algorithm provides near optimal results. More than 90% of the randomly generated problem instances were scheduled with less than 20% overhead compared with the optimal algorithm. The execution time of the scheduling algorithm is two orders of magnitude less than the optimal scheduling algorithm.

5. References

1. J. Babb, R. Tessier, and A. Agarwal, "Virtual Wires: Overcoming Pin Limitations in FPGA-based Logic Emulations," FCCM '93, April 1993.
2. M. R. Garey, D. S. Johnson, and R. Sethi, "The Complexity of Flowshop and Jobshop Scheduling," Mathematics of Operations Research, Vol. 1, No. 2, May 1976.

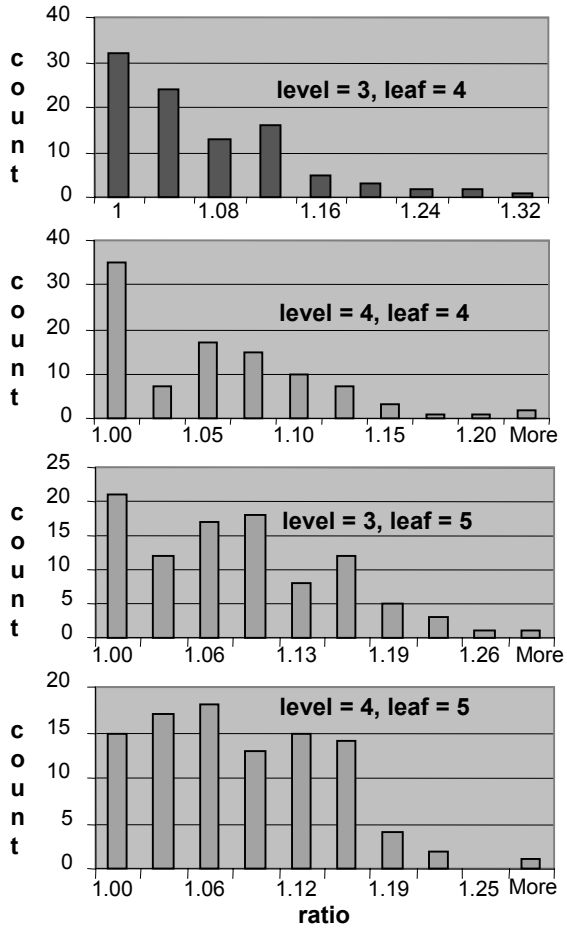


Figure 2. Ratio of the latency produced by our heuristic to optimal latency.

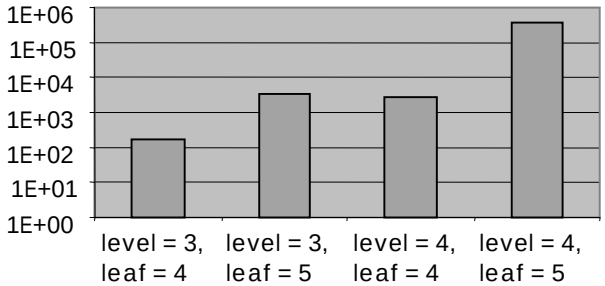


Figure 3. Speedup of the heuristic algorithm over optimal algorithm.

An Efficient Algorithm for Large-Scale Matrix Transposition*

Jinwoo Suh† and Viktor K. Prasanna‡

†USC Information Sciences Institute
4350 N. Fairfax Dr., Suite 770
Arlington, VA 22203
jsuh@isi.edu
http://www.east.isi.edu/~jsuh

‡Department of EE-Systems, EEB-200C
University of Southern California
Los Angeles, CA 90089-2562
prasanna@usc.edu
http://ceng.usc.edu/~prasanna

Abstract

Efficient transposition of large-scale matrices has been widely studied. These efforts have focused on reducing the number of I/O operations. However, in the state-of-the-art architectures, data transfer time and index computation time are also significant components of the overall time. In this paper, we propose an algorithm that considers all these costs and reduces the overall execution time.

The reduction of the overall execution time is achieved by using two techniques: (1) writing the data onto disk in predefined patterns and (2) balancing the numbers of disk read and write operations. Even though our approach may increase the number of I/O operations for some cases it results in an overall reduction in the execution time. The index computation time, which is an expensive operation involving two divisions and a multiplication, is eliminated by partitioning the memory into two buffers. The expensive in-processor permutation is replaced by data collection operations. Our algorithm is analyzed using the well-known Linear Model and the Parallel Disk Model.

The experimental results on a Sun Enterprise and a DEC Alpha show that our algorithm reduces the execution time by about 50%, compared with the best known algorithms in the literature.

*This work was funded by the DoD High Performance Modernization Program ERDC Major Shared Resource Center through Programming Environment and Training (PET) supported by Contract Number DAHC 94-96-C0002, and Subcontract Number NRC-CR-98-0002.

†Most of this work was performed when the first author was a doctoral student at USC. The current work of the author is funded by Defense Advanced Research Projects Agency (DARPA) through the Air Force Research Laboratory, USAF, under agreement number F30602-99-1-0521. Disclaimer: Views, opinions, and/or findings contained in this report are those of the authors and should not be construed as an official Department of Defense, DARPA, or Air Force Research Laboratory position, policy, or decision unless so designated by other official documentation. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon.

1 Introduction

Matrix transpose is a key primitive in a wide variety of scientific computations. In such applications, the typical data size that is stored on the disk is of the order of TeraBytes. To store such data, high-performance computing platforms employ RAID [4] disk systems.

Two models have been widely used in the literature to abstract the behavior of disk systems: the Parallel Disk Model (PDM)[21] and the Linear Model (LM)[14]. The PDM is well suited to model I/O systems such as the RAID [4]. In PDM, the data access cost is represented as $\lceil m/(DB) \rceil \times T_b$, where m is the data size, D is the number of disks, and T_b is time to transfer a block of data (B) between memory and disk. In the Linear Model, the cost is represented as $T_s + m\tau$, where the T_s is startup time, m is data size, and τ is data transfer time per unit data.

In this paper, we propose an efficient algorithm for transposing large-scale matrices (out-of-core matrix transpose). A matrix of size $N \times N$ initially resides on the disk, $N = \prod_{s=0}^{t-1} r_s$, where r_s and t are positive integers. The matrix is to be transposed and stored in another array. The size of the available main memory, M, is smaller than the matrix size. Without loss of generality, we assume that the matrices are stored in row-major order.

Several researchers have studied the out-of-core matrix transpose problem. A straightforward algorithm performs matrix transpose using $O(N^{3/2})$ I/O operations when $M = O(N)$. Eklundh [12] proposed an algorithm that has $O(N \log N)$ I/O complexity assuming $B = N$. Ari et al. [2] modified the algorithm in [12] to reduce the number of I/O operations at the expense of increased number of stages (passes). Floyd [13] derived the upper and lower bounds on the number of I/O operations when $M = 2B$. Aggarwal et al. [1] derived a lower bound on the number of I/O operations

for the general case using PDM. Kaushik et al. [14] reduced the number of I/O operations by a factor of 25% by combining two read operations compared with the algorithm in [12].

All these efforts focus on reducing the number of I/O operations only. However, the main costs in the state-of-the-art architectures consist of not only the time for I/O but also the in-processor data transfer time and index computation time. Figure 1 depicts the breakdown of the various costs in a typical transpose operation.

Our out-of-core matrix transpose algorithm reduces the total execution time by reducing both the number of I/O operations and the index computation time. The reduction in the number of I/O operations is achieved by using efficient data layout on disk and balancing the number of read and write operations. We analyze the complexity of our algorithm using the well-known Parallel Disk Model (PDM) and the Linear Model (LM). A comparison of the algorithms with respect to the number of I/O operations is shown in Table 1.

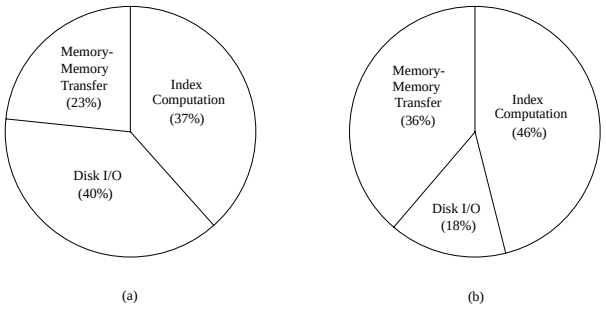


Figure 1: Breakdown of the total execution time for matrix transpose (a) On SGI/Cray T3E (DEC Alpha), $M = 16$ MBytes, data size = 128 MBytes (b) On Sun Enterprise, $M = 64$ MBytes, data size = 2 Gbytes.

To eliminate the index computation cost, our algorithm partitions the available memory into two buffers (read and write buffers). The expensive in-processor permutation is replaced by data collect operations. The write operations and collect operations are scheduled efficiently to reduce the overall time. The size of each buffer is determined by the available memory size and the factorization of N. By using these techniques, the index computation is replaced by inexpensive do-loops (see Section 4.2.2).

We implemented the algorithm on a single node of an SGI/Cray T3E based on DEC Alpha 21164 at the San Diego Supercomputing Center (NPACI/SDSC) and a Sun Enterprise 4000 based on UltraSPARC at the University of Southern California. The experiments were carried out for available main memory sizes ranging from 16 MB to 64 MB and data sizes ranging from 128 MB to 2 GB. The results show that our algorithm reduces the execution time by up to 50%.

The organization of the remainder of this paper is as follows. In Section 2, two well-known disk models are briefly described. In Section 3, previous algorithms for large scale matrix transposition are discussed. Our algorithm is described in detail in Section 4. Experimental results as well as comparisons with previous algorithms are presented in Section 5. Section 6 discusses a further extension of our algorithm to perform Bit-Matrix-Multiply/Complement (BMMC) set of permutations and Section 7 concludes the paper.

2 Disk Models

State-of-the-art disk systems employ sophisticated hardware and perform several optimizations to reduce the I/O time. For example, many of these systems employ a disk buffer, a library buffer, and a controller, and perform access reordering. Each of the above system features needs several parameters to describe its behavior and such a model will be too complex to be useful.

Two models of disk systems that capture the key characteristics of such systems have been widely used in the literature. One of them is the Parallel Disk Model (PDM) [21]. It models the low level behavior of disk systems using several parameters: block size (B) which is the size of data that is transferred between disk and memory in one I/O operation, number of disks (D), memory size (M), number of processors (P), and amount of data transferred (m). The total time for data transfer between disk and memory can be represented as $\lceil m/(DB) \rceil \times T_b$, where T_b is the time to transfer a block of data between memory and disk.

In another model [14], two costs are considered: startup time and data transfer time. The startup time is a fixed time for setting up the data transfer between memory and disk. The rest of the cost is proportional to the amount of data transferred. Thus, it can be represented as $T_s + m\tau$, where T_s is the startup time, m is the data size, and τ is the time to transfer unit data. Typically, T_s is in msec range, and τ is in tens of nsec/byte range.

3 Previous Algorithms

In this section, for the sake of completeness, two well-known algorithms are briefly described. These two algorithms provide the best performance over many other algorithms. The algorithm in [1] has been designed using the PDM and the algorithm in [14] has been designed using the LM. In Section 4, our algorithm is compared with these algorithms.

Table 1: Comparison of the number of I/O operations for D=1(s, $0 \leq s < t$, refers to the stage. See Section 4.2 for details.)

Algorithm	Linear Model (LM)	Parallel Disk Model (PDM)	
		$B \leq \frac{M}{r_s}$	$B > \frac{M}{r_s}$
Aggarwal et al. [1]	-	$\frac{2N^2}{B} \lg_{M/B} \min(\frac{N^2}{B}, B)$	-
Kaushik et al. [14]	$\frac{N^2}{M} \sum_{s=0}^{t-1} (1 + r_s)$	$\frac{2N^2 t}{B}$	$\frac{N^2}{B} \sum_{s=0}^{t-1} (\frac{Br_s}{M} + 1)$
This Paper	$\frac{N^2}{M} \sum_{s=0}^{t-1} \min(r_s, 2(\sqrt{2r_s} + 1))$	$\frac{2N^2}{B} \lg_{M/B} \min(\frac{N^2}{B}, B)$	$\frac{2N^2}{B} \sum_{s=0}^{t-1} (\sqrt{\frac{Br_s}{M}} + \frac{B}{M})$

3.1 Matrix Transpose

In the matrix transpose problem, an input matrix of size $N \times N$ initially resides on the disk, where $N = \prod_{s=0}^{t-1} r_s$, for some $t > 0$, where r_s is a positive integer. If N is a prime number, we can add dummy rows to make N to be nonprime. The input matrix is to be transposed and stored in another array. M, the size of the available memory is smaller than the input matrix size. Throughout this paper, to illustrate the key ideas, we use square matrices. However, the algorithms can be easily extended to rectangular matrices as well, using the technique in [14]. Also, for the sake of simplicity, throughout the paper, we assume that all the ratios are integers.

3.2 Aggarwal's Algorithm

Aggarwal et al. [1] showed a lower bound on the number of I/O operations to perform matrix transpose. In this algorithm, as many blocks as the size of the available memory are read into memory. Then, the data is permuted and written onto the disk. The number of I/O operations for this approach is shown in Table 1.

In this algorithm, r_s is restricted to be $\leq M/B, 0 \leq s < t$. This is because if $r_s > M/B$, a block must be stored $\lceil \frac{r_s}{M/B} \rceil$ times per iteration instead of storing it once. This results in a considerable increase in the number of I/O operations. In our algorithm, we relax this restriction by developing a technique to use a larger block size. Also, this algorithm does not consider index computation time. Index computation is needed to perform permutation of the data in memory.

3.3 Kaushik's Algorithm

In this algorithm [14], there are t stages, where $N = \prod_{s=0}^{t-1} r_s$. Each stage consists of N^2/M steps. In each step, M/N rows are read into memory and a permutation of the data is performed in the memory. Then, the data is written back to the disk in $r_s, 0 \leq s < t$, write operations. Thus, the number of read (write) operations in each step is 1 (r_s). The total number of I/O operations using the LM and the PDM are shown in Table 1.

Although the number of I/O operations and the time to transfer data between memory and disk are considered, the total number of read and write operations are not optimized. Also, the index computation time is not considered.

4 An Efficient Algorithm

We present an overview of our approach in Section 4.1. Section 4.2 provides the details of our approach and analysis using PDM and LM.

4.1 Overview

One of the key features of our algorithm is the reduction in the total number of I/O operations, which is achieved by means of an efficient data layout scheme on the disk. For example, in [14], there are three I/O operations (one read operation and two write operations) in each step when $M = 2N$ and $B = N$. Our algorithm requires only a single write operation in each step as against two write operations in the case of the previous algorithms in [1, 14]. The concept of a step is explained in detail in Section 4.2. Since our algorithm consists of the same number of steps as in the previous algorithms, there is a considerable reduction in the total number of write operations.

This reduction in the number of write operations is a consequence of the efficient data layout scheme $L_s, 0 \leq s < t$, employed by our approach. Note that the initial and final data layouts are the same as in other algorithms. The proposed layout scheme provides a means for reducing the number of write operations while maintaining the same number of read operations. Thus, the number of I/O operations is reduced from three to two in each step which leads to a

33% reduction in the total number of I/O operations.

Another technique used in our algorithm is the balancing the numbers of read and write operations. In balancing the numbers of read and write operations, the key idea is that the total number of I/O operations can be reduced by reducing the number of write operations at the expense of an increased number of read operations. For example, when $r_s = 32$, in each step, the number of read (write) operations in [14] is 1 (32). In our algorithm, we increase the number of read operations to 9 in order to reduce the number of write operations to 9. This results in a 45% reduction in the total number of I/O operations.

As shown in Figure 1 (see page 2 of this paper), the index computation takes up a significant portion of the total execution time. In the previous algorithms [1, 14], the entire available memory is used for reading data from disk. Even though this approach maximizes the memory utilization, it results in excessive index computation cost. (Index computation refers to computing the source or destination addresses of each data.) To eliminate the index computation cost, the available memory is partitioned into two different-sized buffers (read and write buffers). Instead of performing a permutation before every write operation, only the data needed for each write operation is moved into the write buffer. This is denoted as a *collect* operation. The stride of the data access for the collect operation is constant. Thus, it can be performed using inexpensive do-loops.

If the same schedule as in the previous algorithms is used (collect operations followed by write operations), then the size of the write buffer must be $M/2$. However, in our algorithm, the utilization of the write buffer is increased using our schedule which results in a smaller write buffer. In our schedule, a write operation follows each collect operation. Since the read buffer size is less than the available memory size, the number of I/O operations is increased slightly. However, as shown in Section 5, the total execution time is reduced significantly due to reduction in the index computation time.

4.2 Details of the Algorithm

Additional details of our algorithm as well as the analysis are presented in this section. However, due to space limitation, proofs of the theorems are not included. Section 4.2.1 describes our method to reduce the number of I/O operations. It first describes the overall algorithm using the "layout" concept. Then, the layout is explained for four different cases. In section 4.2.2, our method to reduce index computation time is explained.

```
1  for s = 0 to t-1 // for each stage
2      for step = 0 to N^2/M-1 // for each step
3          Read M units of data from disk using
               layout L_{s-1};
4          Permute the data on memory;
5          Write M units of data to disk using
               layout L_s;
```

Figure 2: Overview of the Algorithm

4.2.1 Reducing Number of I/O Operations

Our algorithm to reduce the number of I/O operations is elaborated here (see Figure 2). Note that the matrix size is $N \times N$ and $N = \prod_{s=0}^{t-1} r_s$.

The algorithm consists of t stages. In the s^{th} stage, $0 \leq s < t$, a *submatrix* is defined as follows. Let $d_{i,j}$ denote the data in the row i and column j of the original input matrix. A submatrix $S_{k,l}$, $0 \leq k, l < R_s$, consists of $d_{i,j}, kN/R_s \leq i < (k+1)N/R_s, lN/R_s \leq j < (l+1)N/R_s$, where $R_s = \prod_{i=0}^{s} r_i$.

In each stage, there are N^2/M steps (Line 2). In each step, the data is first read into memory (Line 3). The data in the memory that is in the same submatrix is moved to a contiguous region of the memory (Line 4). A *superblock* denotes such a contiguous region of memory. There are r_s superblocks of size M/r_s. The superblocks are written onto the disk (Line 5). The layout, $L_s, 0 \leq s < t$, specifies where each data is located.

The layout, the schedule for reading data from the disk, and the schedule for writing data onto the disk are explained with the following four cases. Case 1 and Case 2 pertain to the scenarios where as much data as the memory size can be read from the disk or written onto the disk in one I/O operation (i.e., $B = M$). Our analysis shows that efficient data arrangement reduces the number of I/O operations by a factor of $(r_s + 1)/r_s$ (Case 1). In addition to this, if $r_s \geq 8$ (Case 2), balancing the number of I/O operations further reduces the total number of I/O operations.

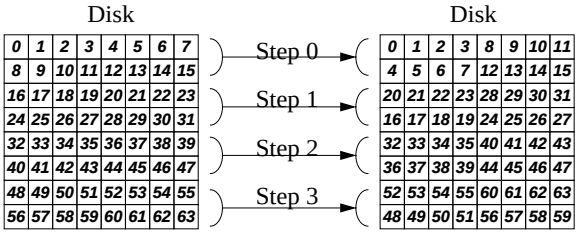
If $M/r_s < B < M$ (Case 3), our algorithm provides the best performance compared with the previous algorithms. Finally, if $B \leq M/r_s$ (Case 4), our algorithm has the same performance as the previous algorithm in [1] with respect to the number of I/O operations.

Note that reducing the index computation time (discussed in Section 4.2.2) further improves the performance in all the cases. In the following, s, $0 \leq s < t$, refers to the stage.

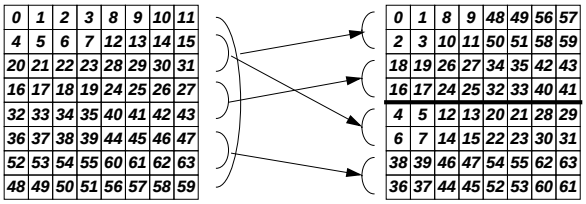
Case 1: ($B = M$ and $1 < r_s < 8$) The key idea here is data arrangement on the disk, L_s. The matrix is first partitioned into R_{s-1} areas. Each area includes

N/R_{s-1} rows.

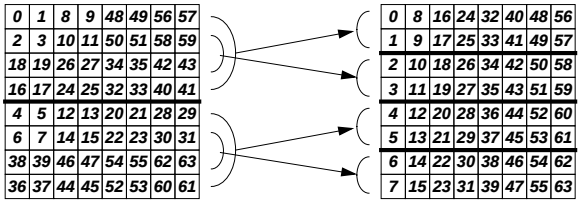
The layout, L_s, for $r_s = 2$, is explained with an example (See Figure 3). The number in each small square denotes a data element. The arrows indicate read and write operations. If two superblocks are adjacent, the data is read or written in one I/O operation. This enables us to write two superblocks in one write operation (for example, see row 0 and row 1 in Figure 3(a)). This also enables us to read two superblocks in one read operation in the next stage (for example, row 1 and row 2 in Figure 3(b)). Similar read and write patterns are repeated for the other superblocks. If two superblocks are adjacent, they can be read or written in one I/O operation (except the first and the last row in each layout). For example, in Figure 3(b), the superblocks in the first and last rows are not adjacent.



(a) Stage 0



(b) Stage 1



(c) Stage 2

Figure 3: An illustrative example($N = 8 = \prod_{s=0}^{2} 2$, and $M = 16$)

If $r_s > 2$, in the st^{th} step, two superblocks, $(st + 1) \bmod r_s$ and $(st + 2) \bmod r_s$, are stored in one write operation and the rest of the data is written in $(r_s - 2)$ write operations which results in $r_s - 1$ write operations.

Case 2: ($B = M$ and $r_s \geq 8$) In this case, the total number of I/O operations can be further reduced by balancing the numbers of read and write operations in addition to the data layout and the schedule explained in Case 1. In Kaushik et al.'s algorithm, the difference between the numbers of read and write operations is large. That is, in each step, the number of read operations is 1 and the number of write operations is r_s. In our algorithm, we develop a technique that reduces the number of write operations at the expense of an increased number of read operations.

Note that a straightforward method reduces the number of write operations to $r_s - z$, where z is the number of the new read operations. Then, the total number of I/O operations is $(r_s - z) + z = r_s$. The total number of I/O operations is reduced by only one. In our algorithm, we decrease the number of write operations to approximately r_s/z. Then, the total number of I/O operations can be reduced by choosing an optimal value of z. In the previous algorithms, each superblock is stored in one disk write operation. In our algorithm, z blocks are stored on the disk in one write operation. Thus, the number of write operations is reduced by a factor of z. In each read operation, to read data that is "scattered" in noncontiguous locations, we need to perform z read operations. It can be shown that, in the s^{th} stage, the optimal value of z is $\sqrt{2r_s}$, $0 \leq s < t$. Theorem 1 applies to Case 1 and Case 2.

Theorem 1 *In the Linear Model, the total number of I/O operations in our algorithm is*

$$\frac{N^2}{M} \sum_{s=0}^{t-1} \min(r_s, \sqrt{2r_s} + 1).$$

Case 3: ($M/r_s \leq B < M$) This is similar to Case 2; the ony difference is the size of the block. It relaxes the restriction ($r_s \leq M/B$) that was imposed in [1]. In our algorithm, we can increase the value of r_s to be larger than M/B so that the number of stages is decreased. The optimal value of z is Br_s/M.

Theorem 2 *In the Parallel Disk Model, the total number of I/O operations in our algorithm is*

$$\frac{2N^2}{M} \sum_{s=0}^{t-1} (\sqrt{\frac{r_s M}{B}} + 1),$$
$$\text{where } \frac{M}{r_s} < B < M,\ 0 \leq s < t.$$

Case 4: ($B \leq M/r_s$) In this case, our algorithm is the same as the algorithm in [1].

4.2.2 Reducing Index Computation Time

In the previous algorithms, the available memory is fully utilized to reduce the number of I/O operations.

In other words, in a read operation, as much data as the size of the memory is read from disk. However, this results in a large index computation time. Permuting the data within the memory requires destination location of each data element to be computed.

To reduce the total execution time, we eliminate the expensive index computation by using the algorithm shown in Figure 4. In our algorithm, we partition the memory into two different-sized buffers: one of size M_r, which is used as a Read buffer and the other of size M_w, which is used as a Write buffer. The read buffer is used for reading data from disk. After reading the data, there are r_s/z sets of collect and write operations, where z is a positive integer (See Section 4.2.2). In each collect operation, data in z submatrices is collected into the write buffer and this operation is repeated r_s/z times (Line 5). The sizes of the write and read buffers are determined as $Mz/(r_s + z)$ and $MR_s/(r_s + z)$, respectively.

In a collect operation, the data in z superblocks is located in $M_r R_{s-1}/N$ chunks of data. The amount of data in each chunk is N/R_{s-1}, where $R_s = \prod_{i=0}^{s} r_i$. Thus, to collect the data into z superblocks, multiple-level do-loops are necessary. In each do-loop, the required computations are simple additions to compute the loop-variables. Note that, in the previous algorithms [1, 14], the computations to permute the data consists of both index computations and loop-variable computations. In our algorithm, since the loop-variables are used to collect data to the write buffer, the index computation is eliminated.

The collected data in the write buffer is written onto the disk in a write operation (Line 6). Even though the number of I/O operations increases by a factor of M/M_r, the total execution time is reduced significantly due to the elimination of index computation time.

5 Experimental Results

We implemented the algorithms on a DEC Alpha system (SGI/Cray T3E, DEC Alpha, 300 MHz) at the San Diego Supercomputing Center (SDSC) and a Sun Enterprise 4000 system (UltraSparc, 336 MHz) at the University of Southern California. For comparison purposes, Kaushik et al.'s algorithm described in Section 3.3 was also implemented.

```
1   for s = 0 to t - 1 // for each stage
2       for step = 0 to N^2/M_r - 1 // for each step
3           Read data from disk;
4           for i = 0 to r_s/z - 1
5                   Move i^th submatrix to
                        write buffer;
6                   Write data in write buffer
                        to disk;
```

Figure 4: Pseudo-code for our algorithm

In our experiments, we observed that Aggarwal et al.'s algorithm, described in Section 3.2, has the same total execution time as the Kaushik et al.'s algorithm. Even though the two algorithms perform the needed permutation using different methods and the permuted data are different, the permutation times are the same. If the block size is smaller than M/r_s in the s^{th} stage, $0 \leq s < t$, then both the algorithms require the same I/O time, where t is the number of stages. I/O time is different for the two algorithms when the amount of data transferred in one I/O operation is smaller than B. The amount of the data transferred in one I/O operation in our experiments ranges from 128 KBytes to 2 MBytes and the typical size of B in state-of-the-art platforms is 4 KBytes. Thus, the performance of the two algorithms is the same in our experiments. Therefore, in Table 2 and Table 3, the execution time reported under the heading "previous" refers to both the algorithms.

The amount of main memory allocated to the data was varied from 16 MBytes to 64 MBytes and the data size was varied from 128 MBytes to 2 GBytes. For each parameter value (memory and data sizes), the algorithms were executed 5 times and the maximum, average, and minimum values were calculated. The reported times are in seconds. The speedup of our algorithm over the previous algorithms was calculated for each parameter setting. The results of our experiments are shown in the Table 2 and Table 3. The results show that our algorithm reduces the execution time by about 50%.

The execution times correlate well with our analysis (as explained in Section 4.2). For example, through our experiments in implementing the algorithms on the Sun Enterprise, we estimated the various parameters that contribute to the overall execution time. Based on this, we arrive at the following empirical equations: 1) Time for reading data from and writing data onto disk = (size of data (in bytes) x number of stages x 2(one for each read and write operation) x data transfer time/byte. 2) Index computation time = size of data x number of stages x index computation time/byte. 3) Data movement time = size of data x number of stages x data movement time/byte. Using these equations we can approximate the overall execution time as the sum of all these contributing factors. Based on our experiments we estimated the following: the time for data transfer = 10 nsec/byte, the time for index computation = 50 nsec/byte and the time for data movement

= 40 nsec/byte. For data size of 2 GBytes and using a 3 stage algorithm, the time for reading data from and writing data onto disk = 2G x 3 stages x 2 x 10 nsec/byte= 128.85 secs, index computation time = 2G x 3 stages x 50 nsec/byte = 322.12 secs and data movement time = 2G x 3 stages x 40 nsec/byte = 257.70 secs. The total execution time = 128.85 + 322.12 + 257.70 = 708.67 secs which is close to the actual values shown in Table 3.

As seen from the tables, the execution time increases by a factor of four when the data size is increased fourfold. This is reasonable since the three major costs (I/O time, index computation time, and memory-memory transfer time) are proportional to the data size. Thus, we can expect similar speedups for data sizes larger than 2 GBytes.

6 Further Extensions

Our matrix transpose algorithm can be extended to the more general problem of performing Bit Matrix Multiply Complement (BMMC) [7] set of permutations.

Performing BMMC consists of several steps. In each step, there are three basic operations as in the case of matrix transpose: read data from disk, permutation of the data in memory, and write data onto disk. We expect our algorithm to improve the overall time to perform BMMC, since in our approach, reducing the number of I/O operations and reducing the index computation time are independent of the permutation of the data in memory.

To reduce the number of I/O operations, the algorithm in Figure 2 is used to employ the two techniques proposed in this paper: efficient data layout and balancing the number of I/O operations.

To reduce the index computation time, the algorithm in Figure 4 is used: the available memory is partitioned into two buffers, the permutation is replaced by collect operations, and the collect operations and write operations are scheduled to maximize the utilization of the available memory.

7 Conclusion

In this paper, we presented an efficient algorithm for large-scale matrix transposition. Contrary to the previous works that have focused only on reducing the number of I/O operations, we identified the major costs in the state-of-the-art computing platforms in performing transpose and the overall cost was reduced. The generality of the main ideas make them applicable to other algorithms having recursive structures that operate on large data sets.

8 Acknowledgements

We would like to thank ERDC and MSRC staff for their support in conducting this work. We would also like to thank Bharani Thiruvengadam and Santosh Narayanan for their assistance in preparing this manuscript.

References

[1] A. Aggarwal and J. S. Vitter, "The Input/Output complexity of sorting and related problems," Communications of the ACM, Vol. 31, No. 9, pp. 1116-1127, 1988.

[2] M. B. Ari, "On transposing large $2^n \times 2^n$ matrices," IEEE Trans. Computers, Vol. C-27, No. 1, pp. 72-75, 1979.

[3] L. Carter, J. Ferrante and S. F. Hummel, "Hierarchical Tiling for Improved Superscalar Performance," Proceedings of IPPS '95, 1995.

[4] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson, "RAID: high performance, reliable secondary storage," ACM Computing Surveys, Vol. 26, No. 2, pp. 145-185, June 1994.

[5] A. Choudhary, W. K. Liao, P. Varshney, D. Weiner, R. Linderman and M. Linderman "Design, Implementation and Evaluation of Parallel Pipelined STAP on Parallel Computers," 12th International Parallel Processing Symposium, Orlando, Florida, 1998.

[6] A. Choudhary, M. Kandemir, H. Nagesh, J. No, X. Shen, V. Taylor, S. More, and R. Thakur, "Data Management for Large-Scale Scientific Computations in High Performance Distributed Systems," in High Performance Distributed Computing Conference '99, San Diego, CA, August 1999.

[7] T. H. Cormen, T. Sundquist, and L. F. Wisniewski, "Asymptotically Tight Bounds for Performing BMMC Permutations on Parallel Disk Systems," Technical Report PCS-TR94-223, Dartmouth College, Department of Computer Science, 1994.

[8] L. G. Delcaro and G. L. Sicuranza, "A method on transposing externally stored matrices," IEEE Trans. on Computers, Vol. C-23, No. 9, pp. 801-803, 1974.

[9] M. Kallahalla and P. Varman, "Optimal Read-Once Parallel Disk Scheduling," Proc. ACM Workshop on I/O in Parallel and Distributed Systems, April 1999.

[10] M. Kallahalla and P. Varman, "An Improved Parallel Prefetching Algorithm," Proc. of Intl. Conference on High Performance Computing, Dec. 1998.

[11] D. E. Dudgen and R. M. Mersereau, Multidimensional Signal Processing, Prentice-Hall, 1984.

[12] J. O. Eklundh, "A fast computer method for matrix transposing," IEEE Transactions on Computers, Vol. 20, Number 7, pp. 801-803, 1972.

[13] R. W. Floyd, "Permuting information in idealized two-level storage, " Complexity of Computer Computations, pp. 105-109, Plenum, 1972.

Table 2: Experimental results on DEC Alpha

Data Size (MBytes)		Memory Size = 16 MBytes			Memory Size = 32 MBytes			Memory Size = 64 MBytes		
		Previous	Our	Speedup	Previous	Our	Speedup	Previous	Our	Speedup
128	Min	45	24	1.88	58	33	1.76	48	32	1.50
	Avg	50	24	2.08	60	36	1.67	48	32	1.50
	Max	53	24	2.21	65	41	1.59	49	33	1.48
512	Min	202	97	2.08	211	126	1.67	197	132	1.49
	Avg	248	125	1.98	235	133	1.77	199	133	1.50
	Max	311	163	1.91	248	141	1.76	201	133	1.51
2048	Min	834	547	1.52	889	452	1.97	841	576	1.46
	Avg	1031	596	1.73	895	479	1.87	849	583	1.46
	Max	1165	667	1.75	906	519	1.75	860	605	1.42

Table 3: Experimental results on SUN Enterprise 4000

Data Size (MBytes)		Memory Size = 16 MBytes			Memory Size = 32 MBytes			Memory Size = 64 MBytes		
		Previous	Our	Speedup	Previous	Our	Speedup	Previous	Our	Speedup
128	Min	36	18	2.00	37	18	2.06	37	18	2.06
	Avg	37	18	2.00	38	19	2.00	38	19	2.00
	Max	37	18	2.06	38	20	1.90	38	20	1.90
512	Min	145	79	1.84	149	82	1.82	147	78	1.88
	Avg	145	80	1.81	150	84	1.79	148	81	1.83
	Max	145	81	1.79	152	86	1.77	149	82	1.82
2048	Min	609	343	1.78	610	335	1.82	630	336	1.88
	Avg	613	353	1.74	613	346	1.77	642	346	1.86
	Max	617	357	1.73	614	355	1.73	653	351	1.86

[14] S. D. Kaushik, C.-H. Huang, J. R. Johnson, R. W. Johnson, and P. Sadayappan, "Efficient Transposition Algorithms for Large Matrices", Supercomputing, 1993.

[15] V. Kumar, A. Grama, A. Gupta, and G. Karypis, Introduction to Parallel Computing, The Benjamin/Cummings Publishing Company, Inc., 1994.

[16] Y. W. Lim, P. B. Bhat, and V. K. Prasanna, "Efficient Algorithms for Block-Cyclic Redistribution of Arrays," 24:298-330, Algorithmica, 1999.

[17] http://www.darpa.mil/ito/research/dis/ index.html, 1999.

[18] H. Park, J. Suh, V. K. Prasanna, and M. Ung, "Parallel Implementation of 2D FFT on High Performance Computing Platforms," DoD HPC User's Conference '98, Houston, Texas, June 1998.

[19] H. K. Ramapriyan, "A generalization of Eklundh's algorithm for transposing large matrices," IEEE Trans. on Computers, Vol. C-24, No. 12, pp. 1221-1226, 1975.

[20] J. Suh and V. K. Prasanna, "Portable Implementation of Real Time Signal Processing Benchmarks on HPC Platforms," International Workshop on Applied Parallel Computing in Large Scale Scientific and Industrial Problems '98, Umea, Sweden, June 1998.

[21] J. S. Vitter and E. A. M. Shriver, "Algorithms for parallel memory I: Two-level memories," Algorithmica, Vo. 12 No. 2-3, pp. 110-147, 1994.

Implementations of Real-time Data Intensive Applications on PIM-based Multiprocessor Systems*

Jinwoo Suh, Ming Zhu[1], Changping Li, Stephen P. Crago, Stephen F. Shank[2], Richard H. Chau[2†], Walter J. Mazur[2], and Rick Pancoast[2]

University of Southern California/Information Sciences Institute
4350 N. Fairfax Drive, Suite 770, Arlington, VA 22203
{jsuh, cli, scrago}@isi.edu

[1]George Mason University
4400 University Drive, Fairfax, VA 22030
mzhu1@gmu.edu

[2]Lockheed Martin Naval Electronics & Surveillance Systems - Moorestown
199 Borton Landing Road M/S 108-210 Moorestown, NJ 08057
{stephen.f.shank, richard.h.chau , walter.j.mazur, rick.pancoast}@lmco.com

Abstract

The growing gap in performance between processor and memory speeds has created a problem for data-intensive applications. A recent approach for solving this problem is to use processor-in-memory (PIM) technology. PIM technology integrates a processor on a DRAM memory chip, which increases bandwidth between the processor and memory. In this paper, we discuss two PIM-based multiprocessor systems, the System Level Intelligent Intensive Computing (SLIIC) Quick Look (QL) board and a hypothetical V-IRAM multiprocessor board. The former system includes eight COTS PIM chips that are connected by a flexible FPGA-based interconnect network. The V-IRAM board modeled contains four Berkeley V-IRAM PIM processors (currently under development) that are connected using FPGAs. The performance of several data intensive applications on the SLIIC QL board is measured. The performance of the applications on the V-IRAM board is modeled at the clock cycle level. The performances of these boards are compared with a PowerPC-based multicomputer and a Pentium system.

1. Introduction

Microprocessor performance has been doubling every 18-24 months, as predicted by Moore's Law, for many years[6]. This performance improvement has not been matched by DRAM (main memory) latencies, which have only improved by 7% per year [6]. This growing gap in performance between processor and memory speeds has created a problem for data-intensive applications.

To solve this problem, many methods have been proposed such as caching and prefetching. However, these methods provide limited performance improvement and can even hinder performance for data-intensive applications. Caching has been the most popular memory latency tolerating technique [12][14]. Caching increases

* Effort sponsored by Defense Advanced Research Projects Agency (DARPA) through the Air Force Research Laboratory, USAF, under agreement number F30602-99-1-0521. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not interpreted as necessarily representing the official policies or endorsement, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), Air Force Research Laboratory, or the U.S. Government.

† Richard Chau was working for Lockheed martin when this work was conducted.

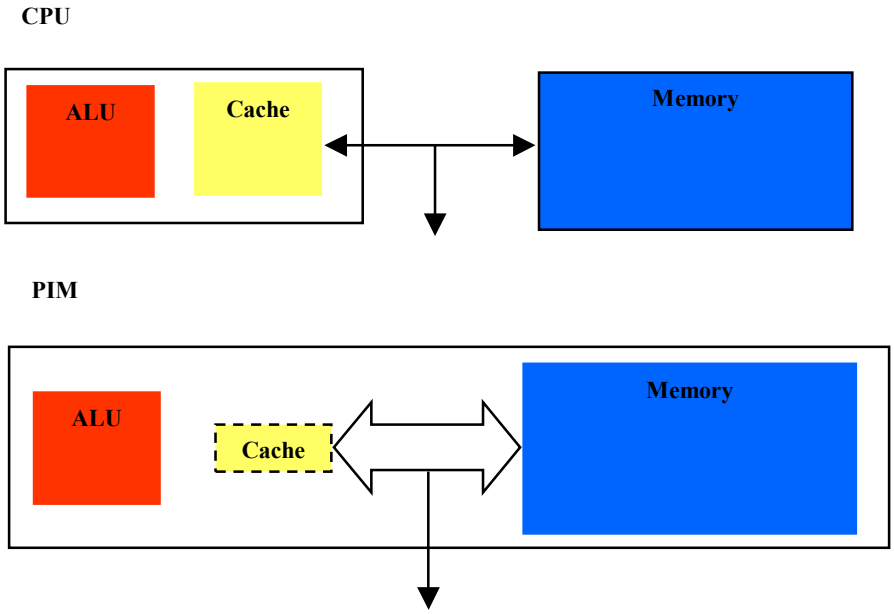


Figure 1. Comparison of PIM and a conventional system

performance by utilizing temporal and spatial locality, but it is not useful for many data-intensive applications such as signal processing applications since they do not show such locality [13].

PIM technology is a promising method for closing the gap between memory speed and processor speed for data intensive applications. PIM technology integrates a processor on a DRAM memory chip. The integration of memory and processor on the same chip has the potential to decrease memory latency and increase the bandwidth between the processor and memory. PIM technology also has the potential to decrease other important system parameters such as power consumption, cost, and area.

To assess existing COTS PIM technology, a System Level Intelligent Intensive Computing (SLIIC) Quick Look (QL) board was implemented and a corner turn (matrix transpose) performance was reported [16]. In this paper, we discuss the implementation of additional data-intensive radar processing applications on the SLIIC QL board: the coherent side-lobe canceller (CSLC) and beam steering. Since the COTS PIM chips were not developed to achieve high performance, we also evaluate the performance of a V-IRAM-based board. The architecture of this board and simulation results are presented in this paper.

The measured and simulated performance is compared with a COTS PowerPC-based multiprocessor and a Pentium III system. It is shown that the performance of the SLIIC QL board is better than a PowerPC-based multiprocessor that consumes more power and occupies more area in data intensive applications. However, it is not known yet weather or not such an improvement can be obtained for general non-data-intensive applications. Simulation results of a V-IRAM multiprocessor board show that the performance will be increased by about 50 times compared with the current SLIIC QL board.

The rest of the paper is organized as follows. In Section 2, PIM technology and PIM chips that are used in the boards evaluated in this paper are briefly discussed. Section 3 describes the architecture of the SLIIC QL board and V-IRAM multiprocessor board. In Section 4, the implementation results of data intensive applications are shown. Section 5 concludes the paper

2. Processor-In-Memory (PIM)

2.1. PIM Technology

Processor-In-Memory (PIM) technology is a promising method for closing the gap between memory speed and processor speed. PIM technology integrates a processor on a DRAM memory chip, which uses DRAM technology. Figure 1 shows a comparison of a PIM and a conventional processor-based system. In the conventional system, the CPU and memory are implemented in different chips. Thus, the bandwidth between CPU and memory is limited since the data must be transferred through chip I/O pins and copper wires on a PCB. In a PIM-based system, the integration of memory and processor on the same chip has the potential to decrease memory latency and increases the bandwidth between the processor and memory. The power usage is smaller than

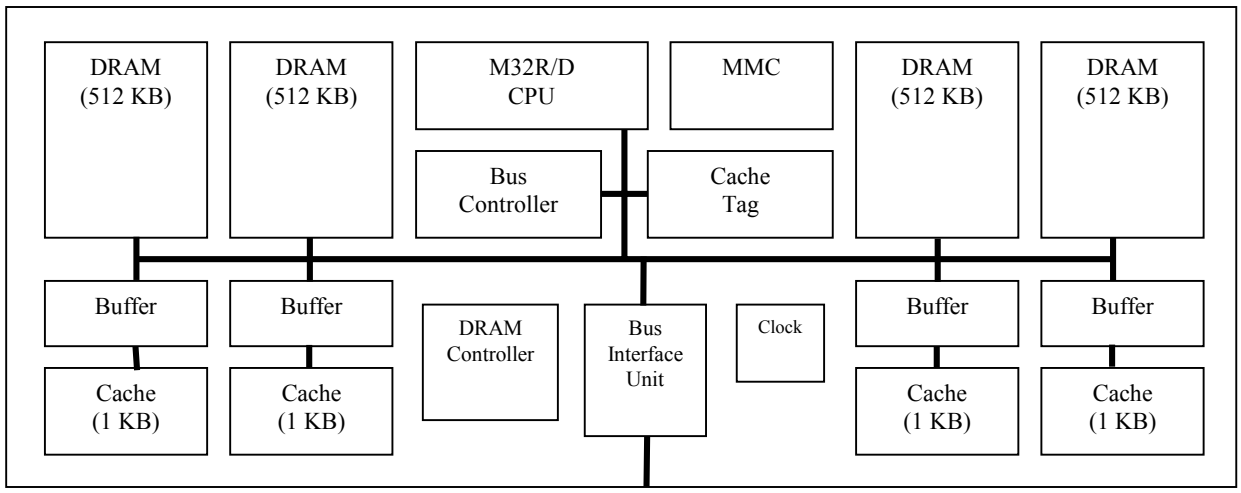


Figure 2. Block diagram of M32R/D

traditional processor-memory chip pair since it takes less power to drive signals within a chip than between chips.

2.2. M32R/D

The only commercial general-purpose PIM chip that has more than 1 MB of DRAM is Mitsubishi M32R/D. The block diagram is shown in Figure 2 [9]. It contains a 32-bit RISC CPU and 2-MByte internal DRAM. Between the CPU and DRAM, there is a 4-KB cache. The bus width between the cache and DRAM is 128 bits. 128 bits of data can be transferred between cache and memory in a clock cycle; however, since the CPU is 32 bits wide, only 32-bit data can be transferred between CPU and cache in a clock cycle. The external clock speed is 20 MHz, and the internal clock speed is 80 MHz. More detail description can be found in [16].

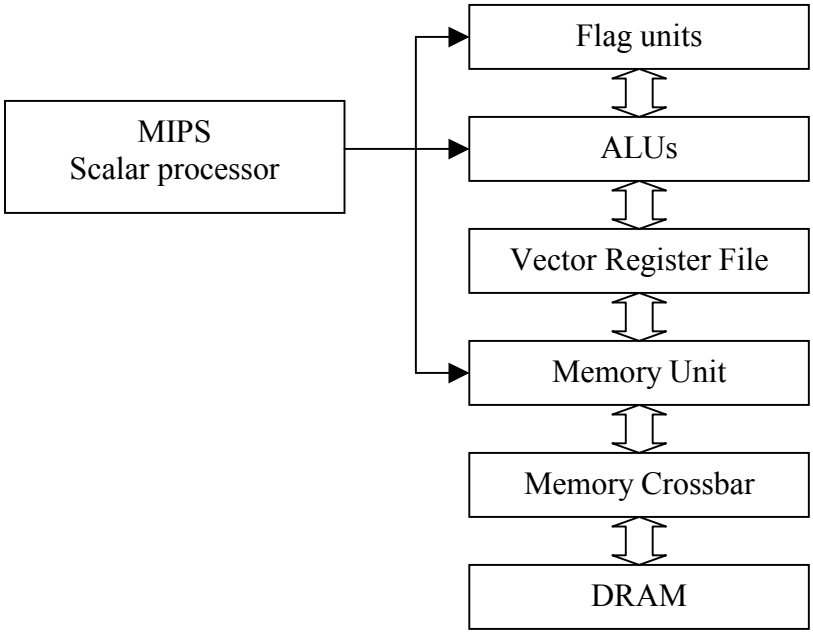


Figure 3. Simplified Block diagram of V-IRAM

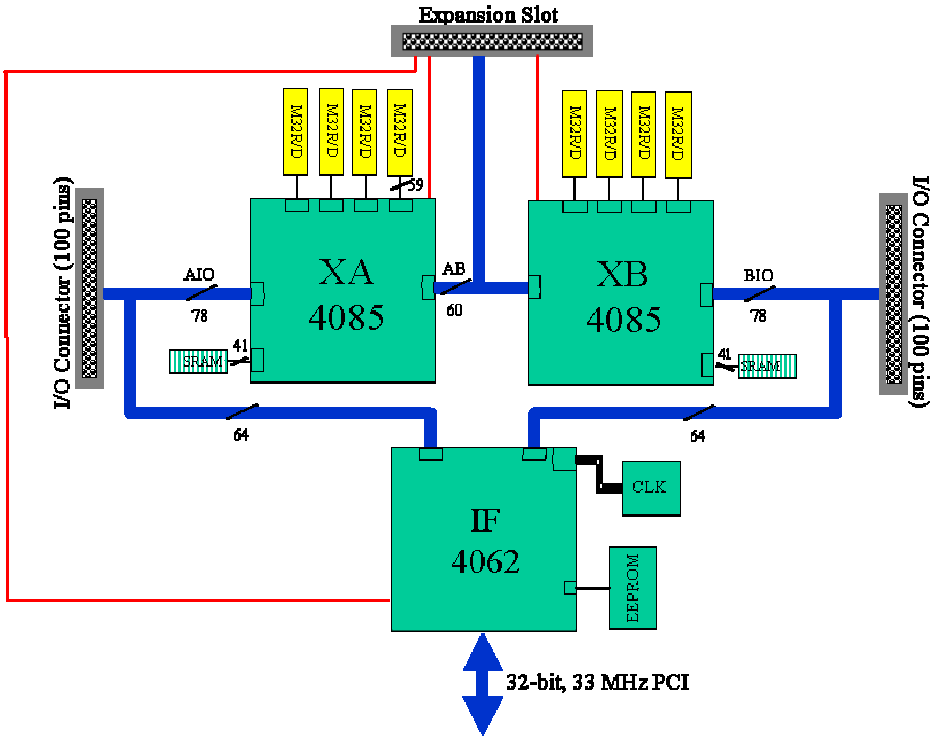


Figure 4. Block diagram of SLIIC QL

2.3. V-IRAM

The V-IRAM chip [7] is a research prototype being developed at the University of California at Berkeley. The simplified architecture of the chip is shown in Figure 3. The V-IRAM contains one vector-processing unit and 8 Mbytes of DRAM in addition to a scalar-processing unit. The vector-processing unit contains two arithmetic units, two flag processing units, and two load/store units. These units are pipelined. Different kinds of operations have different number of stages. The units can be partitioned into several smaller units. For example, it can be partitioned into 4 units each of which performs 64-bit operations. There is a 512-bit data path between the processing units and DRAM. The DRAM is partitioned into two wings, each of which has four banks. There is a crossbar switch between the DRAM and vector processor. The vector processor supports 91 instructions including arithmetic and vector processing. It also supports special vector instructions that help to obtain high performance on dot-product and FFT operations. The target processor speed is 200 MHz, which would provide a peak performance of 3.2 GOPS. The power consumption is expected to be about 2 W.

3. SLIIC QL Architecture

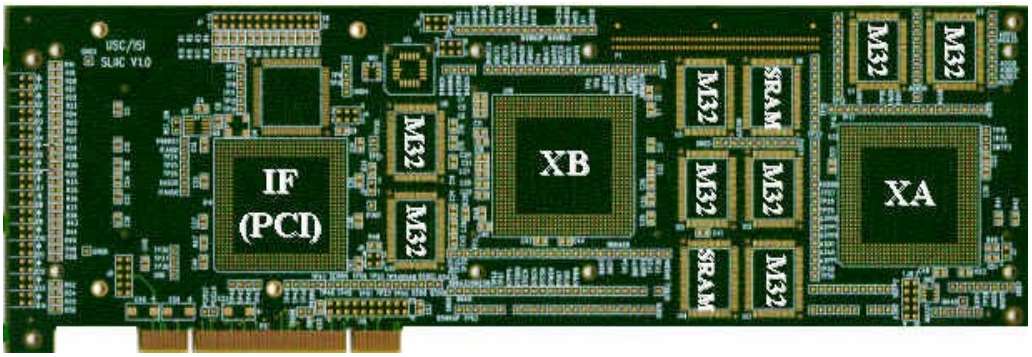


Figure 5. SLIIC QL board photograph

In this section, two PIM architectures are descrbed: SLIIC QL and V-IRAM boards.

3.1. SLIIC QL Board

In this section, a brief description of the SLIIC QL is presented. More detail description of the SLIIC QL board is found in [16] and [17].

A block diagram of the SLIIC Quick Look (QL) board is shown in Figure 4 and a photograph of an unpopulated board is shown in Figure 5. The board fits in a standard PCI form factor, allowing it to be attached to a PC platform. It contains eight Mitsubishi M32R/Ds, providing 640 MIPS of peak processing power and 16 MB of memory.

XA and XB are FPGAs that serve several purposes. First, they provide an interconnection network for the M32R/Ds. Second, XA and XB provide programmable logic that can be used for processor synchronization and performance measurement. Third, the FPGAs provide logic that facilitates communication with the host PC.

The IF FPGA provides the interface to the PCI bus and control for the rest of the board. The user of the SLIIC QL board controls it through a Windows NT-based command line debugger that runs on a PC.

3.2. V-IRAM Multiprocessor Board

The block diagram of the V-IRAM multiprocessor board is similar to SLIIC QL board except that the V-IRAM replaces the M32R/D and one interconnect FPGA is used instead of two. The board contains four V-IRAM PIM processors, one interconnect FPGA for the interconnection network, and one IF FPGA for the interface with a host computer.

The IF FPGA is the same as the IF FPGA on the SLIIC QL board as described in Section 3.1. The internal architecture of the interconnect network logic implements a crossbar switch that connects four PIMs that are connected to the FPGA. Since the number of processors is four, a 5x5 crossbar is implemented. The fifth port connects to IF, through which the PIMs communicate to the host PC.

Communication between two processors is performed through the crossbar. The MIPS scalar processor in the V-IRAM PIM performs communication. Two communication events can be performed simultaneously. A processor cannot receive data from more than one processor simultaneously. The application programmer is responsible for avoiding communication conflicts. There is no hardware or firmware prevention of conflicts. This enables a simpler design, and, as a result, the network is faster.

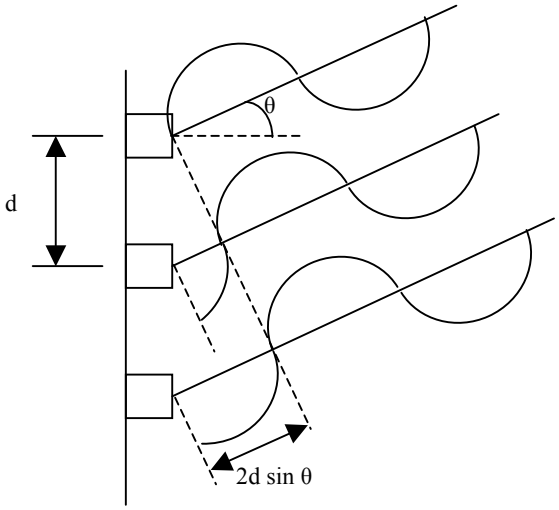


Figure 6. Beam steering

4. Applications

In this section, the two data-intensive applications for which results are presented in this paper are briefly described.

4.1. Beam Steering

Beam steering is a radar processing application that directs a phased-array radar in an arbitrary direction without rotating antenna physically. In a conventional radar system, to send and receive signal from a specific

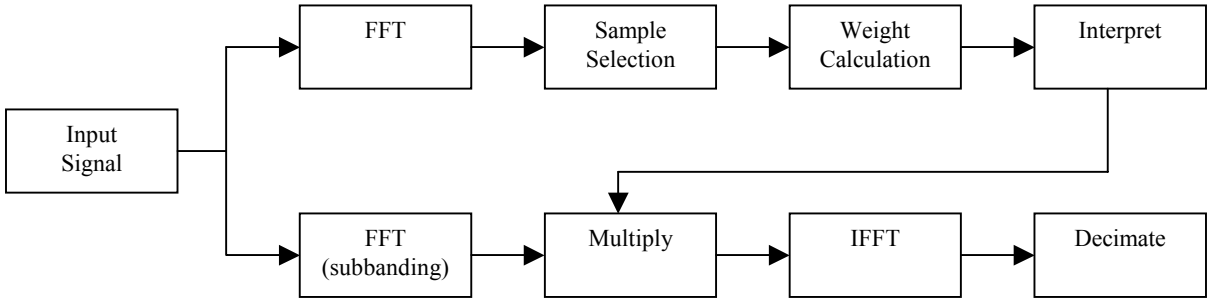


Figure7. Coherent sidelobe canceller (CSLC)

direction, the antenna must be rotated in that direction. This operation needs electrical power to drive a motor and the movement speed is physically limited.

To solve the problem, beam steering is used. Figure 6 shows a one- dimensional beam steering operation. A real system consists of two dimensional antenna elements populated on a plane. In the system, many small antenna elements transmit the signal with different phases. In the figure, each of the three antenna elements transmits a signal with phase shift of d sin θ between adjacent elements. By choosing phases, the antenna direction can be controlled. The computation of the phase for each antenna element involves many load, store and arithmetic operations.

In our implementation, the following parameters are used. The number of antenna elements is 12864 (=17 rows * 17 columns). Each element can direct signal to up to 4 directions per dwell where a dwell is a period. For each direction, the phase needs to be calculated: Depending on the signal frequency and temperature, calibration data needs to be incorporated in the calculation of the phases. In the implementation, four calibration bands are processed.

4.2. Coherent Side-Lobe Canceller (CSLC)

CSLC is a radar signal processing application used to cancel jammer signal caused by one or more jammers. To cancel jammer signals that appear as side-lobes in the frequency domain, one auxiliary channel per jammer signal is needed. The block diagram of the signal processing is shown in Figure 7.

The operations in the upper half of the figure are known as weight calculation and the operations in the lower half are weight application. To cancel the side-lobe, the weight factor is calculated using the signal from the auxiliary channel. Then, the main signal is partitioned into several sub-bands in the time domain. Then, each sub-band is converted to the frequency domain using the FFT (sub-banding). Weight factors are multiplied with the output of the FFT operation to cancel the side-lobe. Then, an inverse FFT is performed on the output data. Most of the computation time is spent on the FFT and IFFT operations. In our implementation, only the weight application is implemented.

The following parameters are used for the implementation. Four input channels, two main channels, and two auxiliary channels are used. Each channel has 8 K data samples. All computations are done using floating-point precision. The data is partitioned into 73 overlapped sub-bands, each of which contains 128 samples. For sub-banding, a 128-sample FFT is used.

5. Experimental Results and Analysis

In this section, the implementation results of the CSLC and beam steering applications are presented. These are implemented or simulated on a SLIIC QL board, V-IRAM multiprocessor board, a PowerPC-based multiprocessor system, and a Pentium III system.

The PowerPC system used for the implementation is a COTS PowerPC-based multiprocessor [1]. The applications are implemented on a CSPI 2741 board, which contains four PowerPC (MPC750, 400 MHz) processors and four Myrinet LaNAI network interfaces.

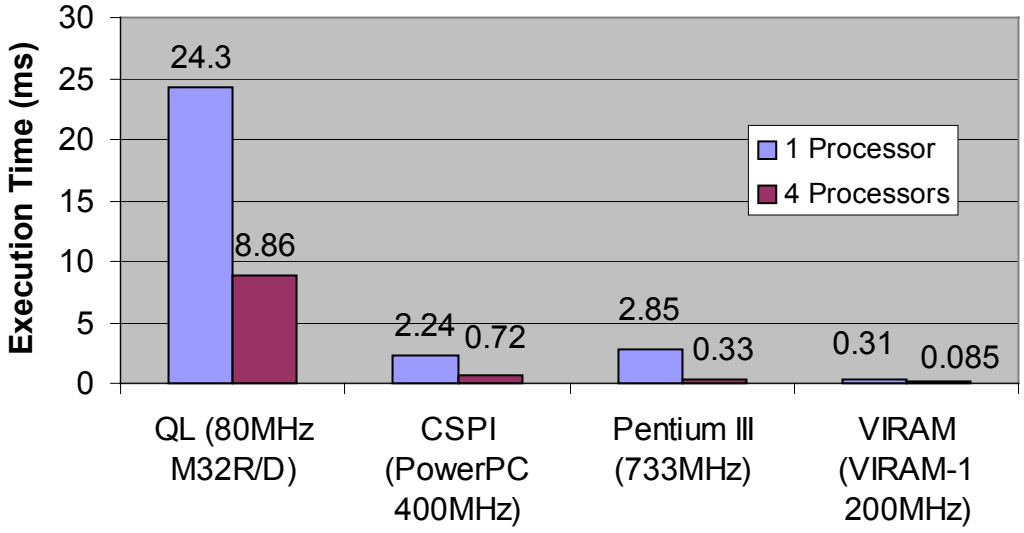


Figure 8. Beam steering implementation results

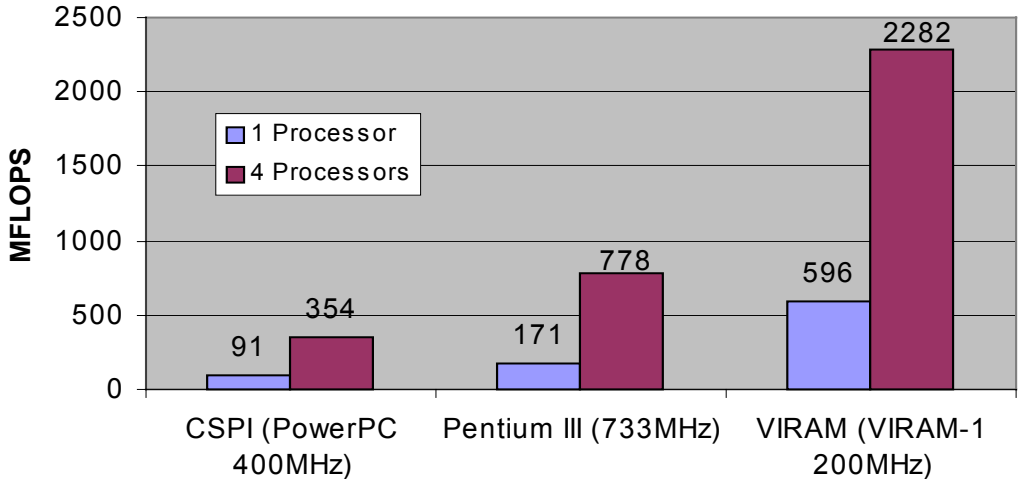


Figure 9. CSLC implementation results

Each node has a 64 Kbyte Level 1 cache and a 1 Mbyte Level 2 cache. The processors are interconnected through a Myrinet network that has eight ports per board of which four ports are connected to the four processors, one is connected to a host system, and the rest are unconnected. The Gcc compiler was used to compile all applications.

For performance measurement on the Pentium III system, a 733 MHz Pentium III-based system was used. We measured only single-processor execution time. The applications were compiled using gcc and the operating system was Linux.

In Figure 8, the implementation results of beam steering are shown. For the PowerPC, Pentium, V-IRAM systems, data sizes are reduced by a factor of four to obtain four-processor system performance. This approach is reasonable since the beam steering (and CSLC) applications can be parallelized in the data domain and the parallelized code does not contain communication among processors.

Note that Pentium III system shows superlinear speedup. We believe this is due to the effect of caching. The graph shows that the V-IRAM multiprocessor provides high speedup compared with other systems even though its clock frequency is not particularly fast. In the other three systems, the performance is roughly proportional to the clock frequency. However, the V-IRAM performance is up to nine times faster than the Pentium III system even though the V-IRAM clock frequency is less than a third of the Pentium III.

In Figure 9, CSLC implementation results are shown. Since the major computation in this application is floating point operations, it was not implemented on SLIIC QL

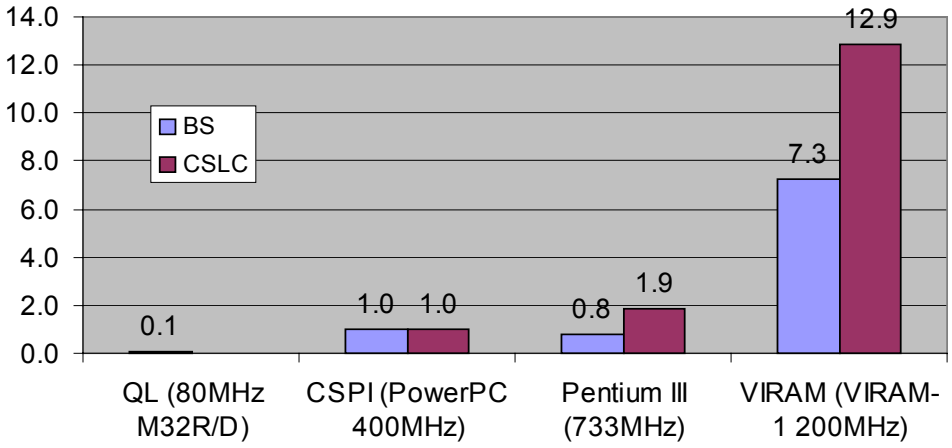


Figure 10. V-IRAM speedup

board as the M32R/D does not support floating-point operations by hardware. In the implantation of the application on the V-IRAM board, the current compiler that is still under development did not optimize the FFT code using the instructions that are suitable for the operations: permutations of data in registers. Thus, we had to extract the FFT core and program in assembly language for that portion. The graph shows that the V-IRAM obtains superior performance on the CSLC application.

A summary speedup graph is shown in Figure 10. The performance on the PowerPC system is shown as a baseline with a speedup of 1. The performance improvement using the V-IRAM board is high due to the wide data path, vector processing unit, and the memory interface unit employed by V-IRAM compared with PowerPC and Pentium. Compared with current SLIIC QL, faster clock speed and latest chip manufacturing technology contributed to the speedup.

6. Conclusions and Future Works

We have presented the implementation results of the real-time data intensive applications, coherent side-lobe canceller and beam steering, on PIM systems and conventional systems. The PIM systems include the SLIIC QL architecture and a hypothetical multiprocessor V-IRAM system. The performance on the V-IRAM system is measured using the V-IRAM performance simulator. The conventional systems evaluated were a PowerPC-based system and a Pentium III-based system. The implementation results show the potential advantages of PIM technology on data intensive applications.

In the future, we are planning to implement a multi-PIM architecture using latest PIM chips. The latest chips will provide higher clock speed, wider data path width, and more parallelism.

7. Acknowledgments

The authors would like to acknowledge Dave Judd, Christoforos Kozyrakis, Kathy Yelick, and the rest of the IRAM team for the use of the V-IRAM compiler and simulator and their generous help.

8. References

[1] CSP Inc., "2000 Series Hardware Manual S2000-HARD-001-01," CSP Inc., 1999.

[2] S. R. Dartt, "Exephere: A Prototype for a Parallel 'Processor-In-Memory' Architecture," Master Thesis, University of Notre Dame, 1988.

[3] DARPA, Data Intensive Systems, http://www.darpa.mil/ito/ research/dis/index.html, 2000.

[4] R. Games, "Benchmarking," http://www.mitre.org/technology/hpc/Data/ct-table.html, 2000.

[5] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, J. Brockman, A. Srivastava, W. Athas, V. Freeh, J. Shin, and J. Park, "Mapping Irregular Applications to DIVA, a PIM-based Data-Intensive Architecture," Supercomputing '99, Portland, OR, November 1999.

[6] J. Hennessy and D. A. Patterson, Computer Architecture: A Quantitative Approach, 2nd Edition, Morgan Kaufmann Publishers, Inc., 1996.

[7] C. Kozyrakis, "A Media-Enhanced Vector Architecture for Embedded Memory Systems," Technical Report # UCB/CSD-99-1059, UC Berkeley, July 1999.

[8] K. Mai, et al, "Smart Memories: A Modular Reconfigurable Architecuture," ISCA 2000, Vancouver, BC, Canada, June, 2000.

[9] Mitsubishi Microcomputers, M32000D4BFP-80 Data Book, http://www.mitsubishichips.com/data/datasheets/mcus/mcupdf/ds/e32r80.pdf.

[10] Motorola, EC603e Embedded RISC Microprocessor Hardware Specifications, http://ebus.mot-sps.com/brdata/PDFDB/MICROPROCESSORS/32_BIT/POWERPC/M951447978093collateral. pdf.

[11] V. K. Prasanna, "Algorithms for Data IntensiVe Applications on Intelligent and Smart MemORies (ADVISOR)," DARPA/ITO Data Intensive Systems Principle Investigator Meeting, May 2000.

[12] S. A. Przybylski, Cache and Memory Hierarchy Design: A Performance-Directed Approach, Morgan Kaufmann Publishers, San Mateo, CA, 1990.

[13] S. Rixner, W. J. Dally, U. J. Kapasi, B. Khailany, A. Lopez-Lagunas, P. R. Mattson, and J. D. Owens, A Bandwidth-Efficient Architecture for Media Processing," 31st Annual International Symposium on Microarchitecture, Dallas, Texas, November 1998.

[14] A. J. Smith, "Cache Memories," Computing Surveys, Vol. 14, No. 3, pp. 473-530, 1982.

[15] M. Snir, "Blue Gene System Overview," Fourth Annual High Performance Embedded Computing Workshop," Boston, MA, September 2000.

[16] J. Suh, S. P. Crago, C. Li, and R. Parker, "A PIM-based Multiprocessor System," International Parallel and Distributed Processing Symposium, San Francisco, CA, 2000.

[17] J. Suh, S. P. Crago, C. Li, S. Shank, R. Chau, W. Mazur, and R. Pancoast, "SLIIC QL Technical Report," USC/ISI Technical Report, In preparation.

[18] Xilinx, http://www.xilinx.com/company/press/kits/pld/fctsheet.htm, 2000.

PIM- and Stream Processor-based Systems*

Jinwoo Suh, Changping Li[1], Stephen P. Crago, and Robert Parker

University of Southern California/Information Sciences Institute
4350 N. Fairfax Drive, Suite 770, Arlington, VA 22203
{jsuh, crago, rparker}@isi.edu

Abstract

The growing gap in performance between processor and memory speeds has created a problem for data-intensive applications. Recent approaches for solving this problem are processor-in-memory (PIM) technology and streaming processor technology.

In this paper, we assess the performance of systems based on PIM and stream processors. The performance of several data-intensive applications are simulated and results are compared with measured performance of conventional systems of systems based on the PowerPC and Pentium processors.

1. Introduction

The growing gap in performance between processor and memory speeds has created a problem for data-intensive applications. PIM technology is a promising method that integrates a processor on a DRAM memory chip. It has the potential to decrease memory latency and to increase the bandwidth between the processor and memory.

The V-IRAM chip [2] is a PIM research prototype being developed at the University of California at Berkeley. The V-IRAM contains one vector-processing unit and 8 Mbytes of DRAM in addition to a scalar-processing unit. There is a 512-bit data path between the processing units and DRAM. The target processor speed is 200 MHz, which would provide a peak performance of 3.2 GOPS.

Another approach for tolerating the performance gap between processor and memory is the stream processor. In this approach, the data is routed through a stream registers to hide memory latency, allow the re-ordering of DRAM accesses, and to minimize the number of accesses.

The Imagine chip [1] is a research prototype streaming processor being developed at Stanford University. It contains eight clusters of arithmetic units that process data from a stream register file. The target processor speed is 500 MHz, which would provide a peak performance of 40 GOPS.

In this paper, we assess the performance of data-intensive radar processing applications on the V-IRAM and Imagine processors. We implemented the beam steering and coherent side-lobe canceller (CSLC) applications and measured performance using cycle-level accurate simulators. We show that the performance of systems based on these processors is better than PowerPC and Pentium-based systems for these applications.

2. Applications

2.1. Corner Turn

The out-of-place corner turn is a matrix transpose in which two matrices are used: source and destination matrices. The matrix size used for this paper, which was chosen to be larger than most caches, is 1024 x 1024 with 4-byte elements.

2.2. Beam Steering

Beam steering is a radar processing application that directs a phase-array radar in an arbitrary direction without physically rotating the antenna. The antenna consists of a two-dimensional array of antenna elements populated on a plane. In the system, each antenna elements can transmit its signal with a different phase. The computation of the phase for each antenna element involves many load, store and arithmetic operations.

In the implementation used for this paper, the following parameters are used. The number of antenna elements is 12,864 (17 rows and 17 columns). The array of antenna elements can direct signal to up to 4 directions per dwell (period). For each direction, the phase needs to be calculated. Depending on the signal frequency and temperature, calibration data needs to be incorporated in the calculation of the phases. In this implementation, four calibration bands that are used to adjust parameters based on ambient environment such as temperature are processed.

2.3. Coherent Side-Lobe Canceller (CSLC)

CSLC is a radar signal processing application used to

* Effort sponsored by Defense Advanced Research Projects Agency (DARPA) through the Air Force Research Laboratory, USAF, under agreement number F30602-99-1-0521. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not interpreted as necessarily representing the official policies or endorsement, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), Air Force Research Laboratory, or the U.S. Government.

[1] Changping Li was an employee of USC/ISI when this work was performed.

cancel jammer signals caused by one or more jammers that appear as side-lobes in the frequency domain. To cancel the side-lobe, weight factors are generated using the signal from the auxiliary channel. Then, the main signal is partitioned into several sub-bands in the time domain, and each sub-band is converted to the frequency domain using an FFT (sub-banding). Weight factors are multiplied with the output of the FFT operation to cancel the side-lobe. Finally, an inverse FFT is performed to convert the output data back to the time domain. In our implementation, the weight generation is not implemented.

The following parameters are used for the implementation used in this paper. Four input channels, two main channels, and two auxiliary channels are assumed. Each channel has 8,192 data samples. All computations are done using floating-point precision. The data is partitioned into 73 overlapped sub-bands, each of which contains 128 samples. For sub-banding, a 128-sample FFT is used.

3. Experimental Results and Analysis

In this section, the implementation results of the corner turn, CSLC, and beam steering applications are presented. Performance of these applications is estimated using cycle-accurate simulators provided by the V-IRAM and Imagine teams. For comparison purposes, actual measurements of application performance were taken using a single node of a PowerPC-based multiprocessor system and a Pentium III system.

Table 1. Implementation results

	Corner Turn (MB/sec)	CSLC (msec)	Beam Steering (msec)
PPC G3 (400 MHz)	21.0	16.6	3.76
Pentium III (733 MHz)	83.9	32.2	4.74
V-IRAM (200 MHz)	1441.8	2.57	0.31
Imagine (500 MHz)	1199.1	0.77	0.30

In Table 1, the implementation results of corner turn, beam steering, and CSLC are shown. The graph of Figure 1 shows that V-IRAM and Imagine provide speedups upto 70 compared with a PowerPC system even though their clock frequencies are not particularly fast.

The results show that the V-IRAM performs better than Imagine on a corner turn, which fits in its main memory. This is due to the fact that V-IRAM has higher bandwidth between memory (which is on-chip on V-IRAM and off-chip on Imagine) and the processing unit. However, Imagine has higher computational performance, which is reflected in the performance of the CSLC, which is more computation-intensive. V-IRAM and Imagine have similar performance on the beam-steering application because of the balance between memory lookups and computation of that application.

In a real system, the current implementation of the V-IRAM may take less space than the current implementation of the Imagine since it has a scalar processor and internal DRAM on-chip and does not need external memory if the application fits in the memory. The Imagine chip includes a network interface and router, which reduces chip count in multiprocessor systems.

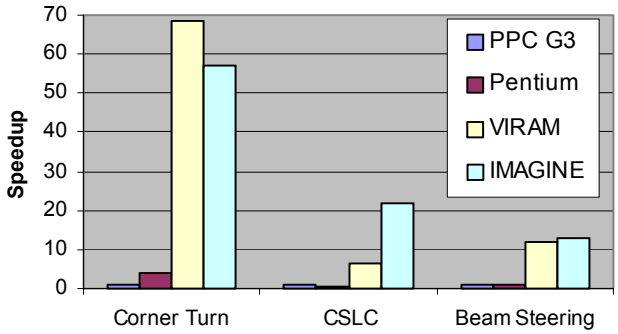


Figure 1. Speedup

4. Conclusions

We have presented simulated performance results for data-intensive radar processing applications on systems based on the V-IRAM PIM and the Imagine streaming processor and compared them to conventional systems. The results show the potential advantages of the new technologies on data intensive applications.

5. Acknowledgement

The authors gratefully acknowledge the UC Berkeley IRAM team and the Stanford Imagine team for the use of their compilers and simulators and their generous help. The authors also acknowledge Rick Pancoast, Steve Shank, Walt Mazur, and Joe Racosky of Lockheed Martin NE & SS for providing the applications.

6. References

[1] B. Khailany, et. al., "Imagine: Signal and Image Processing Using Streams". HOT Chips 12, Stanford, CA, August 2000.

[2] C. Kozyrakis, “A Media-Enhanced Vector Architecture for Embedded Memory Systems,” Technical Report # UCB/CSD-99-1059, UC Berkeley, July 1999.

A PIM-based Multiprocessor System*

Jinwoo Suh, Changping Li, Stephen P. Crago, and Robert Parker

University of Southern California/Information Sciences Institute
4350 N. Fairfax Drive, Suite 770, Arlington, VA 22203
{jsuh, cli, crago, rparker}@isi.edu

Abstract

The growing gap in performance between processor and memory speeds has created a problem for data-intensive applications. A recent approach for solving this problem is to use processor-in-memory (PIM) technology. PIM technology integrates a processor on a DRAM memory chip, which increases bandwidth between the processor and memory. In this paper, we discuss a PIM-based multiprocessor system, the System Level Intelligent Intensive Computing (SLIIC) Quick Look (QL) board. This system includes eight COTS PIM chips and two FPGA chips that implement a flexible interconnect network. The performance of the SLIIC QL board is measured and analyzed for the distributed corner-turn application. We show that the performance of the current SLIIC Ql on the distributed corner turn application is better than a PowerPC-based multicomputer that consumes more power and occupies more area. This advantage, which can be achieved in a limited context, demonstrates that even limited COTS PIMs have some advantages for data-intensive computations.

1. Introduction

Microprocessor performance has been doubling every 18-24 months, as predicted by Moore's Law, for many years [9]. This performance improvement has not been matched by DRAM (main memory) latencies, which have only improved by 7% per year [9]. This growing gap in performance between processor and memory speeds has created a problem for data-intensive applications.

To solve this problem, many methods have been proposed. However, these methods provide limited performance improvement or even lower performance for data intensive applications. Among them, the cache has been the most popular method[1][16][18]. Since caches increase performance by utilizing temporal and spatial localities, they are not useful for data intensive applications such as signal processing applications since they do not show such localities[17].

Another method used to tolerate memory latency is data prefetching[6][8]. By reading data before it is necessary, prefetching can hide the memory-cache latency. However, the method can only be applied to programs with certain access patterns. Also, it is limited by the size of cache and the size of working set.

Processor-In-Memory (PIM) technology is a promising method for closing the gap between memory speed and processor speed for data intensive applications. PIM technology integrates a processor on a chip that uses DRAM technology. The integration of memory and processor on the same chip has the potential to decrease memory latency and increase the bandwidth between the processor and memory. PIM technology also has the potential to decrease other important system parameters such as power consumption, cost, and area.

In this paper, we discuss the implementation of a multiprocessor board that uses PIMs and a reconfigurable interconnect that uses FPGAs. The SLIIC Quick Look board was designed to facilitate the investigation of the possible benefits of PIMs by using current COTS PIM technology.

The SLIIC QL board is implemented as an attached compute accelerator and is implemented on a PCI board, allowing it to be hosted by a PC and other common desktop computers. The main components of the SLIIC QL board are eight PIM chips connected by FPGAs. The programmable logic of the FPGAs allows for a flexible interconnection between the processors, host control interface, and performance measurement.

The performance of the SLIIC QL board is measured and analyzed using a distributed corner turn (matrix

* Effort sponsored by Defense Advanced Research Projects Agency (DARPA) through the Air Force Research Laboratory, USAF, under agreement number F30602-99-1-0521. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not interpreted as necessarily representing the official policies or endorsement, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), Air Force Research Laboratory, or the U.S. Government.

transpose) application. The performance is compared with a COTS PowerPC-based multiprocessor. It is shown that the performance of the SLIIC QL board is better than a PowerPC-based multiprocessor that consumes more power and occupies more area in data intensive applications. However, it is not expected to have such an improvement for general computation-intensive applications.

The rest of the paper is organized as follows. In Section 2, PIM technology and the related work are briefly discussed. Section 3 describes the architecture of the SLIIC QL board. In Section 4, the corner turn implementation results are shown. Section 5 presents an analysis of the corner turn performance. Section 5 concludes the paper.

2. Processor-In-Memory (PIM) Technology and Related Work

PIM technology allows a wide bus between processor and memory that allows fast data transfer between the two components. The power usage is smaller than traditional processor-memory chip pair since it takes less power to drive signals within a chip than between chips.

The only commercial PIM chip that has more than 1 MB of DRAM is Mitsubishi M32R/D [12]. It contains a 32-bit CPU and 2-MByte internal DRAM. The CPU is a five stage RISC and has 83 instructions. Most instructions are executed in one clock. No floating-point operation is supported.

The DRAM is partitioned into four blocks. For fast data access, there is a 4-KB cache. If data is available on cache, the data is accessed from cache; otherwise the data is transferred from memory to cache first, and then transferred to CPU. The cache line size is 128 bits. The cache can be disabled by the user. The bus width between the cache and DRAM is 128 bits.

I/O is performed through the Bus Interface Unit, which transfers 16 bits using an external clock. The external clock speed is 20 MHz, and the internal clock speed is 80 MHz.

The PIM is operated in one of the three modes: active, sleep, and stand-by modes. In active mode, all components are active. In sleep mode, only memory is active and external device can write to or read from the memory. In stand-by mode, all components are disabled; however, the contents of the memory are preserved.

There are several PIM research prototypes being developed. One of them is the V-IRAM chip [10]. The V-IRAM contains two 256-bit vector-processing units and 8 Mbytes of DRAM in addition to a scalar-processing unit. Another PIM chip under development is the DIVA chip [7], which has 8 MB of DRAM and 1024 bits of total data path width.

The Exesphere project implemented a multi-PIM architecture [3]. Each Exesphere board contains nine M32R/D processors. There is a bus that supports only one communication access at a time between the processors through which processors communicate. The Exesphere board is dependent on a separate board for the PCI interface. Even though simulation results are reported, no actual application execution results have been reported. Our work provides real execution results on PIM chips.

The IBM Blue Gene project is also investigating the use of PIMs in a high-performance parallel architecture [19]. The Blue Gene architecture is being developed for modeling the folding of human proteins. The Blue Gene project is developing multithreaded PIM processors.

The ADVISOR project has begun to investigate the algorithmic framework on the design of applications for PIM architectures[15]. The ADVISOR framework models the main characteristics of PIM chips and then, algorithms are designed based on the model. Example algorithms are reported for various applications.

3. SLIIC QL Architecture

3.1. Hardware

The SLIIC QL board uses eight Mitsubishi M32R/D processors. The M32R/D processor was chosen because it provides a relatively large memory size (2 MB) and high data path width between memory and cache (128 bits). It also has a small footprint that enables many processors to fit on a board.

A block diagram of the SLIIC Quick Look (QL) board is shown in Figure 1. The board fits on a standard PCI form factor and contains eight Mitsubishi M32R/Ds, providing 640 MIPS of peak processing power and 16 MB of memory.

XA and XB are FPGAs that serve several purposes. First, they provide an interconnection network for the M32R/Ds. Second, XA and XB provide programmable logic that can be used for processor synchronization and performance measurement. Each FPGA has an attached SRAM memory that can be used to store tables of performance counters and data. Third, the FPGAs provide logic that facilitates communication with the host PC.

There are two advantages to implement these functions in FPGAs. First, modification of the functions and circuits can be done without changing the hardware. The FPGA configurations are specified using the VHDL hardware description language. This is very important for research work where many changes are inevitable. Second, the flexibility allows the board to be used for other applications. When the board is used for another application, the logic in the FPGAs is changed to accommodate new needs.

3.2. Firmware

The internal architecture of the interconnect network logic implemented with the interconnect FPGAs is described in this section.

The two interconnection FPGAs, XA and XB, implement the same logic. Each FPGA implements a

crossbar switch that connects four PIMs that are connected to the FPGA. The crossbar is pipelined to increase throughput. Figure 2 shows the interconnection architecture.

In each FPGA, there is a 7x7 crossbar switch. Four crossbar inputs and outputs are connected to the four processors in the cluster (where a cluster means four processors connected to an FPGA). When communication is performed among processors in a cluster, the 4x4 sub-crossbar is used. To communicate with a processor in a different cluster, one of the two paths between the two clusters must be used. To use a path, the processors use one of the two ports connected to the two paths in the 7x7 crossbar. The remaining port (omitted from Figure 2) connects to IF through which PIMs communicate to host PC.

Communication between two processors is implemented as follows. To communicate another processor, a processor initiates the communication by setting a path between the two processors. Then, the source processor sends data to the destination processor. After communication, the source processor resets the path. A processor cannot receive data from more than one processor simultaneously. The application programmer is responsible for avoiding communication conflicts. There is no hardware or firmware prevention of conflicts. This enables a simpler design, and, as a result, the network is faster.

Each processor has its internal memory (address 0 to 0x1F FFFF) and remote memory mapped to its address 0x20 0000 to 0x3F FFFF. All remote memory (memory on other PIM chips) is mapped to the same address space. A destination ID number sent before a message determines the actual PIM that is mapped to the external memory address range at any given time. For intra-cluster communication, the crossbar pipeline has three stages.

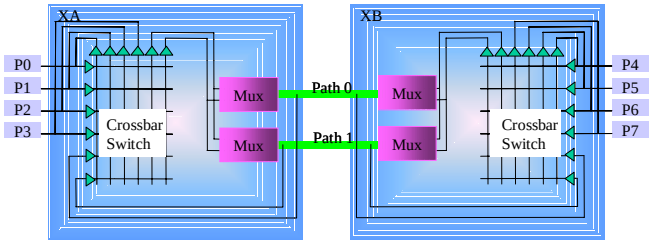


Figure 2. Interconnect network architecture

The two FPGAs clusters communicate through two bi-directional paths (unidirectional at a given time). The application programmer or system software needs to set the path number to use in addition to the destination processor ID before sending a message. Inter-cluster communication has five pipeline stages.

For processor synchronization, barrier synchronization is supported. Barrier synchronization is implemented using FPGA logic. Barrier synchronization can be performed with very little overhead (a few system clock cycles).

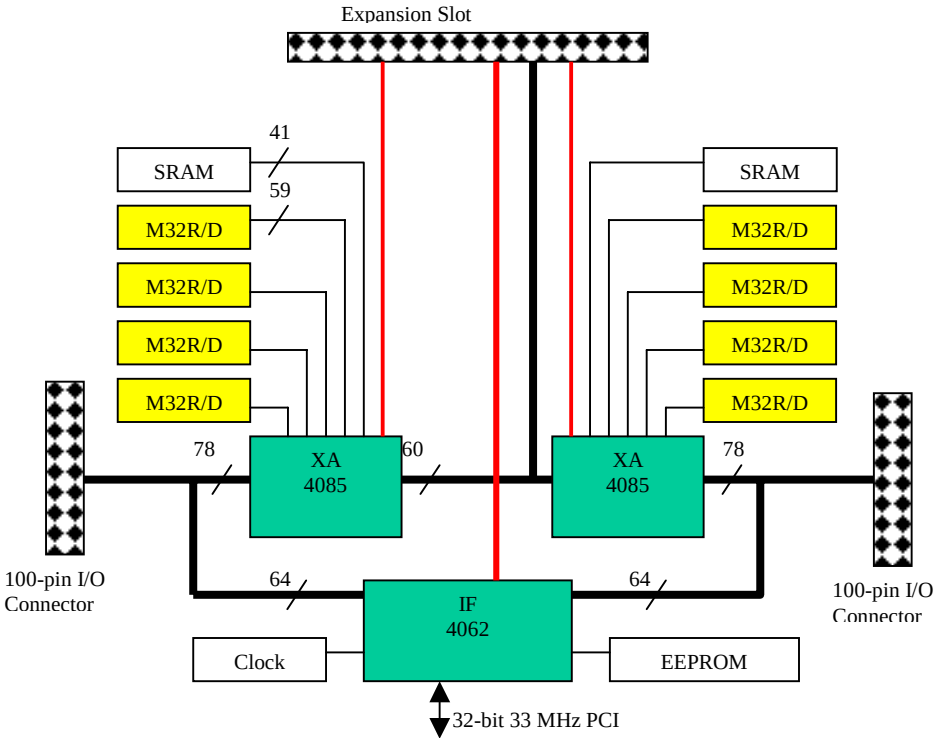


Figure 1. SLIIC architecture

3.3. Software

The SLIIC QL board communicates with the host PC using the SLIIC debugger. The SLIIC debugger, which runs on Windows NT, allows a user to start execution, read data from PIMs, write data to PIMs, read counter data (there is a counter associated with each processor that is used for measurement of execution time), etc. On start-up, the debugger initializes itself and automatically sets the SLIIC QL board clock speed to 20 MHz. More detailed information, such as supported commands and memory address mapping can be found in [20].

For application programmers, an application programming interface (API)

is provided that includes a communication interface, barrier synchronization, and timer control. More detailed information, such as supported commands and memory address mapping can be found in [20].

4. Experimental Results and Analysis

In this section, the performance of the SLIIC QL board is reported using the corner turn (matrix transpose) application. Then, these experimental results are analyzed.

Corner turn is a matrix transpose operation. Initially, a matrix is stored in memory. Then, the matrix is transposed and stored in the same matrix or another matrix. If more than one processor are involves, then, communication cost can be a major cost. Also involved is local transpose cost based on the communication paradigm.

In this paper, various kinds of corner turn performance were measured: local in-place, local out-of-place, intra-cluster in-place, intra-cluster out-of-place, and inter-cluster out-of-place corner turns. In a local corner turn, only one processor was used to perform the corner turn, and in an intra-cluster corner turn, the four processors connected to a single FPGA were used. In an inter-cluster corner turn, all eight processors on the board were used. Local corner turns were executed on a single processor to measure the on-chip bandwidth. Distributed corner turns were executed to measure the memory to I/O bandwidth.

In the in-place corner turn, the source and destination matrices were the same, and in out-of-place corner turn, they were distinct. For all multiprocessor corner turns, all processors served as both source and destination. Figure 3 shows the measured corner turn performance on the QL board. All corner turn code was written in C and compiled with gcc using optimization level –O2.

For local corner turns, in-place performs better than out-of-place. This is because fewer address calculations are necessary for the in-place corner turn. Address calculations are re-used for the source and destination data. For the distributed corner turns (intra-cluster), out-of-place performs better because an extra data transfer to a temporary buffer is required for the in-place case. In the local case, this temporary buffering is done in the processor registers, but the larger message sizes of the distributed corner turn require an extra memory transfer.

The performance of the intra-cluster out-of-place corner turn is almost double that of the one-processor performance. A doubling of performance is the maximum that can be expected because two pairs of processors are exchanging data at any given time. The level of performance achieved indicates that the I/O capability of the M32R/D does not limit performance severely.

The performance drops significantly when the data size in a processor is equal to or greater than 4 KB, which is the size of the cache on the M32R/D. This phenomenon was not observed in local out-of-place corner turns, indicating that the performance of the local out-of-place corner turn is not limited by the cache. Possible bottlenecks are the address computation and bandwidth of the system bus, which connects the processor, cache, and DRAM. The system bus also is used for I/O, but I/O is not used in the local corner turns. The low performance at the small matrix sizes is due to loop overhead.

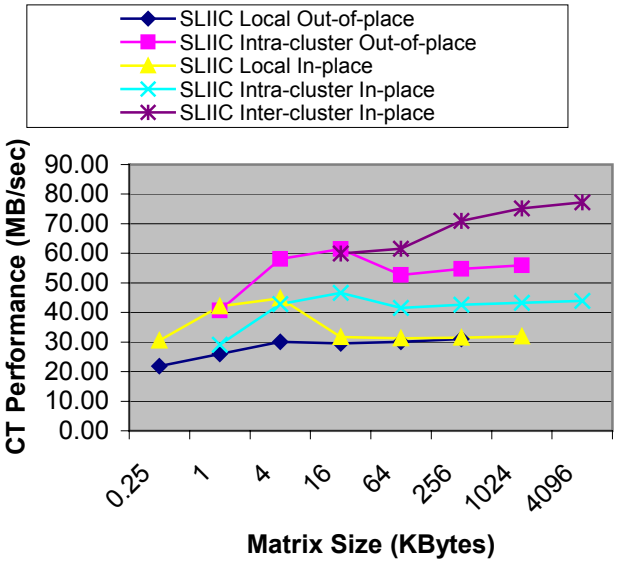


Figure 3. QL Corner Turn Performance

The performance of the out-of-place corner turn with eight processors when the data size is 4 MB is 77.2 MB/sec, which is approximately 40% more than the performance of four-processors. It is not twice the performance of four processors because of the inter-cluster paths limit performance. To fully utilize the eight processors during a corner turn, there must be four paths for inter-cluster communication. Only two paths are available in the current network configuration. Thus, some of the processors idle during the corner turn, which degrades the performance.

For comparison purpose, corner turn operations were also implemented on a COTS PowerPC-based multiprocessor, the CSPI 2641 [1]. Each CSPI 2641 consists of two boards. Each board contains two PowerPC (200 MHz) processors and a LaNAI network interface. The processors are interconnected through a Myrinet network. In our system, since only one board in each CSPI 2641 was available, two CSPI 2641s were used to use four processors. The gcc compiler using –O2 and MPI library were used. Note that the corner turn on CSPI needs local data movement before and after the communication due to the use of MPI library that does not provide stride access.

Figure 4 shows the performance of the QL board compared to the CSPI 2641. The graph shows the performance of a distributed corner turn running on four processors and local corner turn performance. The performance here is defined as the matrix size per execution time.

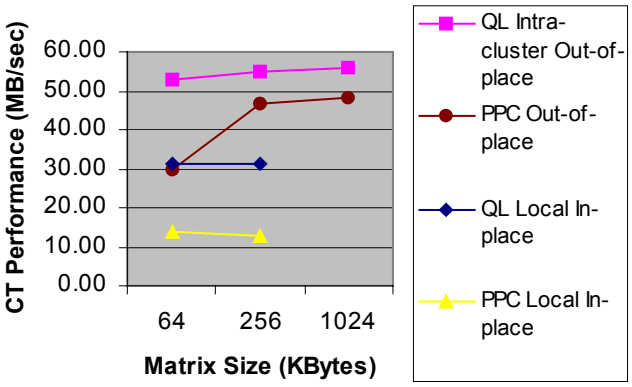


Figure 4. Distributed Corner Turn Performance Comparison with PowerPC

Similar results of out-of-place corner turn on Mercury multicomputer system using PowerPC were reported [5]. The graph shows that the performance of the QL board is better than the PowerPC-based platform.

200 MHz PPC is a little outdated since the latest PPC is 733 MHz now (MPC7450). However, for comparison purpose, since the M32R/D is 80 MHz, 200 MHz is a reasonable comparison when we employ a rule of thumb that the processor speed is two to three times faster than DRAM technology.

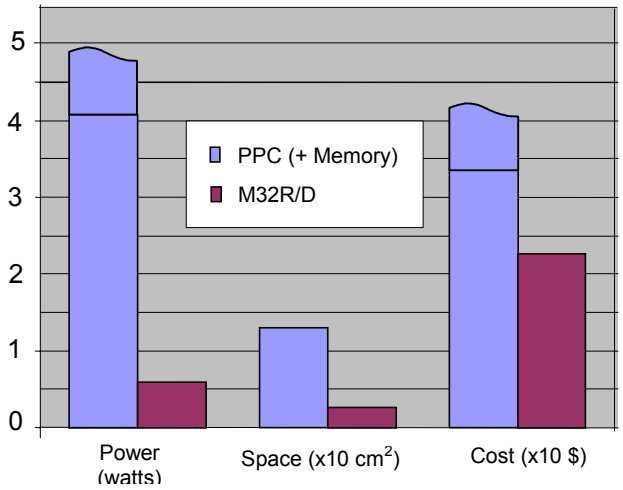


Figure 5. Comparison with PowerPC

The area and power consumed are much different as shown in Figure 5. The four-processor implementation of the corner turn on the QL board runs on five chips (four M32R/Ds and one FPGA for interconnect). The chip count for the COTS platform is at least triple, with each of the four nodes containing a processor, DRAM, and network interface. The power advantage for the M32R/D processors is even more pronounced. Each M32R/D

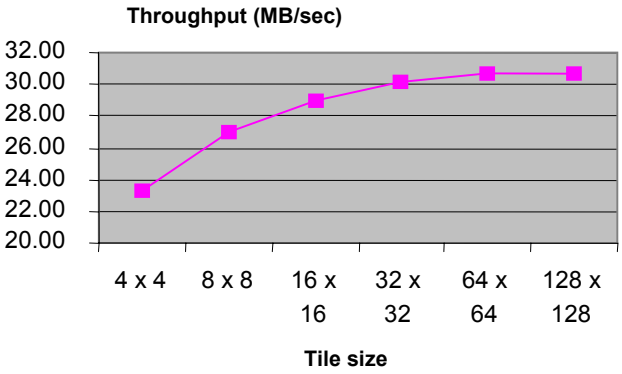


Figure 6. Performance versus tile size

typically consumes 0.54 W [12], while each PowerPC consumes 4.2 W [10]. While the PowerPC-based platform does have advantages, such as higher computational throughput and a more robust message-passing implementation, this comparison shows the potential advantages that PIM processors have for applications that require high bandwidth memory accesses in a small footprint and with low power consumption. The power and cost comparison is shown in Figure 5.

To understand the effect of the cache on the corner turn using PIM, the performance of out-of-place local corner turn using tiling is measured. The tiling is used to maximize the cache utilization. The matrix size was 128 x 128. Tile size is varied from 4 x 4 to 128 x 128.

Using this method, there was no performance improvement compared with non-tiling corner turn. This indicates that the cache is not bottleneck on the M32R/D when data size is large. If the block size is small, the overhead incurred by more address computation reduces the overall performance significantly (up to 30%). When the block size is large, that effectively reduce overhead and it improves the performance compared with small block size.

5. Analysis

In this section, the analysis of the corner turn performance is presented. The analysis is for the data size of 16 bytes since it is the cache line size.

To read 16 bytes of data (a cache line size on an M32R/D), the number of required cycles are [12]:

- 5 cycles to move the first word from DRAM to cache (This also brings three additional word together),
- 3 cycles (one per word for the three extra words),
- 0.25 cycles for page miss (a page miss every 512 Bytes, penalty = 8 cycles),
- 0.13 cycles for refresh (a refresh takes 8 cycles is performed every 512 cycles), and
- 0.0625 cycles for branch (assuming 16 instructions per loop and a loop causes 2 stalls).

-

Thus, reading 16 bytes takes 8.4425 clock cycles, and the maximum performance is 151.6 MB/sec. Our best experimental result was 129.7 MB/sec (85.5% of theoretical maximum).

To write 16 bytes of data, the required cycles are:

- 8 cycles to move data from DRAM to cache (previous data must be sent to DRAM first),
- 8 cycles (two per one word),
- 0.25 cycles for page miss (a page miss every 512 Bytes, penalty = 8 cycles),
- 0.13 cycles for refresh (a fresh takes 8 cycles, and a refresh every 512 cycles), and
- 0.0625 cycles for branch (assuming 16 instructions per loop and a loop causes 2 stalls).

Thus, writing 16 bytes takes 16.4425 and the maximum performance is 77.85 MB/sec. Our best experimental result was 72.7 MB/sec (93% of theoretical maximum).

Corner turn performance can be analyzed using the results of the read and write performance.

For 16 bytes of data, the required cycles are:

- 8.4425 (for load) and 16.4425 (for store) cycles are needed as described before,
- 2 cycles for address calculations,
- 1 cycles for loop variable,
- 1 cycle for compare, and
- 3 cycles for branch.

The total is 31.885 cycles which provides 38.28 MB/sec for a local out-of-place corner turn. The experimental result is 31.88 MB/sec (83.3 % of theoretical maximum).

6. Conclusions and Future Works

We have described the SLIIC QL architecture, a single-board PIM-based multiprocessor with a programmable interconnect. We have shown that the board performs a distributed corner turn with performance better than a COTS PowerPC-based multicomputer using less area and power in data intensive applications.

In the future, we are planning to implement a multi-PIM architecture using the latest PIM chips, which will provide higher clock speed, wider data path width, and more parallelism.

7. References

[1] C. Conti, D. H. Gibson, and S. H. Pitowsky, “Structural aspects of the System/360 Model 85, Part I: General Organization,” IBM Systems Journal, Vol. 7, No. 1, pp. 2-14, 1968.

[2] CSP Inc., “2000 Series Hardware Manual S2000-HARD-001-01,” CSP Inc., 1999.

[3] S. R. Dartt, “Exephere: A Prototype for a Parallel ‘Processor-In-Memory’ Architecture,” Master Thesis, University of Notre Dame, 1988.

[4] DARPA, Data Intensive Systems, http://www.darpa.mil/ito /research/dis/index.html, 2000.

[5] R. Games, “Benchmarking,” http://www.mitre.org/ technology/hpc/Data/ct-table.html, 2000.

[6] A. Gupta, J. L. Hennessy, K. Gharachorloo, T. Mowry, and W. D. Weber, “Computative Evaluation of Latency Reducing and Tolerating Techniques,” Proc. 18th Annual International Symposium on Computer Architecture, Toronto, May 1991.

[7] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, J. Brockman, A. Srivastava, W. Athas, V. Freeh, J. Shin, and J. Park, “Mapping Irregular Applications to DIVA, a PIM-based Data-Intensive Architecture,” Supercomputing '99, Portland, OR, November, 1999.

[8] K. Hwang, Advanced Computer Architecture Parallelism Scalability Programmability, McGraw-Hill, 1993.

[9] J. Hennessy and D. A. Patterson, Computer Architecture: A Quantitative Approach, 2nd Edition, Morgan Kaufmann Publishers, Inc., 1996.

[10] C. Kozyrakis, “A Media-Enhanced Vector Architecture for Embedded Memory Systems,” Technical Report # UCB/CSD-99-1059, UC Berkeley, July 1999.

[11] K. Mai, et al, “Smart Memories: A Modular Reconfigurable Architecuture,” ISCA 2000, Vancouver, BC, Canada, June,June 2000.

[12] Mitsubishi Microcomputers, M32000D4BFP-80 Data Book, http://www.mitsubishichips.com/data/datasheets/mcus/ mcupdf/ds/e32r80.pdf.

[13] Motorola, EC603e Embedded RISC Microprocessor Hardware Specifications, http://ebus.mot-sps. com/brdata/PDFDB/MICROPROCESSORS/32_BIT/POWERP C/M951447978093collateral. pdf.

[14] D. A. Patterson, J. L. Hennessy, Computer organization and design: the hardware/software interface, Morgan Kaufmann, 1994.

[15] V. K. Prasanna, “Algorithms for Data IntensiVe Applications on Intelligent and Smart MemORies (ADVISOR),” DARPA/ITO Data Intensive Systems Principle Investigator Meeting, May 2000.

[16] S. A. Przybylski, Cache and Memory Hierarchy Design: A Performance-Directed Approach, Morgan Kaufmann Publishers, San Mateo, CA, 1990.

[17] S. Rixner, W. J. Dally, U. J. Kapasi, B. Khailany, A. Lopez-Lagunas, P. R. Mattson, and J. D. Owens, “A Bandwidth-Efficient Architecture for Media Processing,” 31st Annual International Symposium on Microarchitecture, Dallas, Texas, November 1998.

[18] A. J. Smith, “Cache Memories,” Computing Surveys, Vol. 14, No. 3, pp. 473-530, 1982.

[19] M. Snir, “Blue Gene System Overview,” Fourth Annual High Performance Embedded Computing Workshop,” Boston, MA, September 2000.

[20] J. Suh, S. P. Crago, C. Li, S. Shank, R. Chau, W. Mazur, and R. Pancoast, “SLIIC QL Technical Report,” USC/ISI Technical Report, In preparation.

[21] Xilinx, http://www.xilinx.com/company/press/kits/pld/ fctsheet.htm, 2000.

PIM- and Stream Processor-based Systems*

Jinwoo Suh, Changping Li[1], Stephen P. Crago, and Robert Parker

University of Southern California/Information Sciences Institute
4350 N. Fairfax Drive, Suite 770, Arlington, VA 22203
{jsuh, crago, rparker}@isi.edu

Abstract

The growing gap in performance between processor and memory speeds has created a problem for data-intensive applications. Recent approaches for solving this problem are processor-in-memory (PIM) technology and streaming processor technology.

In this paper, we assess the performance of systems based on PIM and stream processors. The performance of several data-intensive applications are simulated and results are compared with measured performance of conventional systems of systems based on the PowerPC and Pentium processors.

1. Introduction

The growing gap in performance between processor and memory speeds has created a problem for data-intensive applications. PIM technology is a promising method that integrates a processor on a DRAM memory chip. It has the potential to decrease memory latency and to increase the bandwidth between the processor and memory.

The V-IRAM chip [2] is a PIM research prototype being developed at the University of California at Berkeley. The V-IRAM contains one vector-processing unit and 8 Mbytes of DRAM in addition to a scalar-processing unit. There is a 512-bit data path between the processing units and DRAM. The target processor speed is 200 MHz, which would provide a peak performance of 3.2 GOPS.

Another approach for tolerating the performance gap between processor and memory is the stream processor. In this approach, the data is routed through a stream registers to hide memory latency, allow the re-ordering of DRAM accesses, and to minimize the number of accesses.

The Imagine chip [1] is a research prototype streaming processor being developed at Stanford University. It contains eight clusters of arithmetic units that process data from a stream register file. The target processor speed is 500 MHz, which would provide a peak performance of 40 GOPS.

In this paper, we assess the performance of data-intensive radar processing applications on the V-IRAM and Imagine processors. We implemented the beam steering and coherent side-lobe canceller (CSLC) applications and measured performance using cycle-level accurate simulators. We show that the performance of systems based on these processors is better than PowerPC and Pentium-based systems for these applications.

2. Applications

2.1. Corner Turn

The out-of-place corner turn is a matrix transpose in which two matrices are used: source and destination matrices. The matrix size used for this paper, which was chosen to be larger than most caches, is 1024 x 1024 with 4-byte elements.

2.2. Beam Steering

Beam steering is a radar processing application that directs a phase-array radar in an arbitrary direction without physically rotating the antenna. The antenna consists of a two-dimensional array of antenna elements populated on a plane. In the system, each antenna elements can transmit its signal with a different phase. The computation of the phase for each antenna element involves many load, store and arithmetic operations.

In the implementation used for this paper, the following parameters are used. The number of antenna elements is 12,864 (17 rows and 17 columns). The array of antenna elements can direct signal to up to 4 directions per dwell (period). For each direction, the phase needs to be calculated. Depending on the signal frequency and temperature, calibration data needs to be incorporated in the calculation of the phases. In this implementation, four calibration bands that are used to adjust parameters based on ambient environment such as temperature are processed.

2.3. Coherent Side-Lobe Canceller (CSLC)

CSLC is a radar signal processing application used to

* Effort sponsored by Defense Advanced Research Projects Agency (DARPA) through the Air Force Research Laboratory, USAF, under agreement number F30602-99-1-0521. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not interpreted as necessarily representing the official policies or endorsement, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), Air Force Research Laboratory, or the U.S. Government.

[1] Changping Li was an employee of USC/ISI when this work was performed.

cancel jammer signals caused by one or more jammers that appear as side-lobes in the frequency domain. To cancel the side-lobe, weight factors are generated using the signal from the auxiliary channel. Then, the main signal is partitioned into several sub-bands in the time domain, and each sub-band is converted to the frequency domain using an FFT (sub-banding). Weight factors are multiplied with the output of the FFT operation to cancel the side-lobe. Finally, an inverse FFT is performed to convert the output data back to the time domain. In our implementation, the weight generation is not implemented.

The following parameters are used for the implementation used in this paper. Four input channels, two main channels, and two auxiliary channels are assumed. Each channel has 8,192 data samples. All computations are done using floating-point precision. The data is partitioned into 73 overlapped sub-bands, each of which contains 128 samples. For sub-banding, a 128-sample FFT is used.

3. Experimental Results and Analysis

In this section, the implementation results of the corner turn, CSLC, and beam steering applications are presented. Performance of these applications is estimated using cycle-accurate simulators provided by the V-IRAM and Imagine teams. For comparison purposes, actual measurements of application performance were taken using a single node of a PowerPC-based multiprocessor system and a Pentium III system.

Table 1. Implementation results

	Corner Turn (MB/sec)	CSLC (msec)	Beam Steering (msec)
PPC G3 (400 MHz)	21.0	16.6	3.76
Pentium III (733 MHz)	83.9	32.2	4.74
V-IRAM (200 MHz)	1441.8	2.57	0.31
Imagine (500 MHz)	1199.1	0.77	0.30

In Table 1, the implementation results of corner turn, beam steering, and CSLC are shown. The graph of Figure 1 shows that V-IRAM and Imagine provide speedups upto 70 compared with a PowerPC system even though their clock frequencies are not particularly fast.

The results show that the V-IRAM performs better than Imagine on a corner turn, which fits in its main memory. This is due to the fact that V-IRAM has higher bandwidth between memory (which is on-chip on V-IRAM and off-chip on Imagine) and the processing unit. However, Imagine has higher computational performance, which is reflected in the performance of the CSLC, which is more computation-intensive. V-IRAM and Imagine have similar performance on the beam-steering application because of the balance between memory lookups and computation of that application.

In a real system, the current implementation of the V-IRAM may take less space than the current implementation of the Imagine since it has a scalar processor and internal DRAM on-chip and does not need external memory if the application fits in the memory. The Imagine chip includes a network interface and router, which reduces chip count in multiprocessor systems.

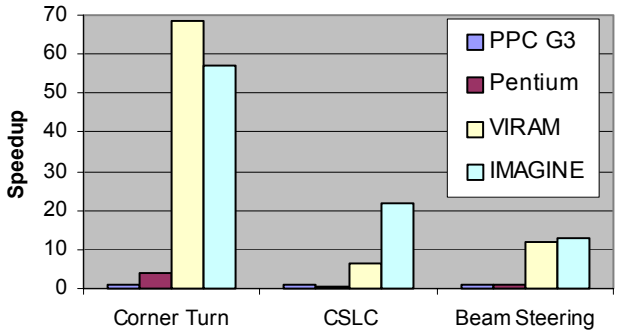


Figure 1. Speedup

4. Conclusions

We have presented simulated performance results for data-intensive radar processing applications on systems based on the V-IRAM PIM and the Imagine streaming processor and compared them to conventional systems. The results show the potential advantages of the new technologies on data intensive applications.

5. Acknowledgement

The authors gratefully acknowledge the UC Berkeley IRAM team and the Stanford Imagine team for the use of their compilers and simulators and their generous help. The authors also acknowledge Rick Pancoast, Steve Shank, Walt Mazur, and Joe Racosky of Lockheed Martin NE & SS for providing the applications.

6. References

[1] B. Khailany, et. al., "Imagine: Signal and Image Processing Using Streams". HOT Chips 12, Stanford, CA, August 2000.

[2] C. Kozyrakis, “A Media-Enhanced Vector Architecture for Embedded Memory Systems,” Technical Report # UCB/CSD-99-1059, UC Berkeley, July 1999.

Efficient Algorithms for Fixed-Point Arithmetic Operations In An Embedded PIM*

Jinwoo Suh, Dong-In Kang, and Stephen P. Crago

University of Southern California/Information Sciences Institute
4350 N. Fairfax Drive, Suite 770, Arlington, VA 22203
{jsuh, dkang, crago}@isi.edu

Abstract

The growing gap in performance between processor and memory speeds has created a problem for data-intensive applications. A recent approach for solving this problem is to use processor-in-memory (PIM) technology. PIM technology integrates a processor on a DRAM memory chip, which increases bandwidth between the processor and memory.

An example of a PIM is the Mitsubishi M32R/D, which has a processor and 2 MB of DRAM on a chip. Even though the chip has many advantages, it has limited support for arbitrary fixed-point multiplication and division, which are necessary for some embedded computing applications. A software implementation of arbitrary fixed-point arithmetic operation is required for these applications. Straightforward implementations lose either precision or performance. Thus, algorithms that are fast and accurate are needed.

In this paper, we design efficient algorithms for fixed-point arithmetic that use integer arithmetic. The algorithms are analyzed and implementation results on a PIM-based multiprocessor system, the System Level Intelligent Intensive Computing (SLIIC) Quick Look (QL) board, are presented.

Keywords: fixed-point arithmetic, algorithm, PIM, multiplication, division

1. Introduction

Microprocessor performance has been doubling every 18-24 months, as predicted by Moore's Law, for many years [5]. This performance improvement has not been matched by DRAM (main memory) latencies, which have only improved by 7% per year [5]. This growing gap in performance between processor and memory speeds has created a problem for data-intensive applications.

To solve this problem, many methods have been proposed, including caching [5][10][12] and data prefetching [3]. However, these methods provide limited performance improvement or even lower performance for data-intensive applications since these applications do not show the localities required by these techniques [11].

Processor-In-Memory (PIM) technology is a promising method for closing the gap between memory speed and processor speed for data intensive applications. PIM technology integrates a processor on a chip that uses DRAM technology. The integration of memory and processor on the same chip has the potential to decrease memory latency and increase the bandwidth between the processor and memory. PIM technology also has the potential to decrease other important system parameters such as power consumption, cost, and area.

* Effort sponsored by Defense Advanced Research Projects Agency (DARPA) through the Air Force Research Laboratory, USAF, under agreement number F30602-99-1-0521. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not interpreted as necessarily representing the official policies or endorsement, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), Air Force Research Laboratory, or the U.S. Government.

Several research PIM chips are being developed [2][4][6]. These provide very high performance compared with general-purpose processors [14][15].

An example of a commercial PIM is the Mitsubishi M32R/D, which operates at 80 MHz and contains 2 MB of internal DRAM [8]. Even though the M32R/D has many advantages, its processing capability is somewhat limited since it was originally targeted to low-power applications that do not require high performance, such as digital cameras and personal digital assistant (PDA) devices. One limitation of the M32R/D is that non-integer fixed-point multiplication and division are not supported. In embedded computing, however, it is sometimes necessary to calculate data in an arbitrary fixed-point format.

For example, the coherent side-lobe canceller (CSLC) implemented at USC/ISI requires fixed-point operations since the input and output data are in fixed-point formats [15]. CSLC is a radar signal processing application that cancels jammer signals. In this implementation, all data is in a fixed-point format with a 21-bit integer and an 11-bit fraction.

Thus, a software implementation of fixed-point arithmetic is required for the CSLC application. Since the M32R/D does not support floating-point processing either, the fixed-point operations must be implemented using integer operations.

For fixed-point addition and subtraction, straightforward implementations exist and can be used. However, for multiplication and division, precision is lost if a straightforward implementation (e.g. [1]) is used. When this method is used on the M32R/D and other processors without special support, precision is lost in the shift operation used during the computation.

To preserve precision, another straightforward method converts data to a double-precision format. Since the double-precision format preserves all data bits, precision is not lost. However, this approach involves many additional fixed-point operations and additional register space.

In this paper, we discuss efficient algorithms for fixed-point arithmetic operations for the M32R/D and other general-purpose processors that have limited or no support for arbitrary fixed-point arithmetic operations. Compared with a straightforward algorithm, our algorithm reduces the number of arithmetic operations significantly. The algorithm is analyzed and implementation results on a System Level Intelligent Intensive Computing (SLIIC) Quick Look (QL) board [13], which is a multiprocessor board that uses eight M32R/D chips, are presented.

The rest of the paper is organized as follows. In Section 2, the problem and previous approaches are briefly discussed. Section 3 describes our algorithms, and Section 4 describes the architecture of the SLIIC QL board on which the experiments were performed. Section 4 also contains the algorithm implementation results. Section 5 concludes the paper.

2. Problem and Previous Approaches

The M32R/D supports only integer format arithmetic operations. To represent arbitrary fixed-point data, we conceptually put a decimal point at position p, $0 \leq p \leq 32$. For example, if $p = 0$, then the data is integer and if $p = 32$, then all data is less than 1. Throughout this paper, p indicates the position of the decimal point.

For addition and subtraction, it is straightforward to implement fixed-point arithmetic operations with integer arithmetic. Integer operations are performed and the results are valid because the decimal point does not shift in these operations.

However, for multiplication, the position of the decimal point is shifted. Consider two numbers, a and b. The product of a and b is $a * b$ with the decimal point at $2p$. Thus, when the arithmetic operations are performed using integer hardware, it is necessary to adjust the position of the decimal point, and the result must be shifted to the right by p positions.

Figure 1 shows the correct multiplication of the fixed-point numbers 1.0 and 1.0. In the figure, p is assumed to be 11. The left block indicates the integer portion of a number and the right block indicates the fraction. Thus, 1.0 is indicated by a one followed by 11 zeros, and the multiplication result should be 1 followed by 11 zeros. However, if the operands are multiplied as integers, the hardware multiplies 4096 by 4096 and the result is 16777216, which is a one followed by 22 zeros in binary representation (see Figure 2).

Therefore, to get a result with the correct decimal point position, the data must be right shifted. A straightforward method of the multiplication [1] is shown in Figure 3. Let us denote sr(a, b) and sl(a, b) as a b-position shift-right and shift-left of number a, respectively. The shift right operation causes the p leftmost bits of data to be zero when and the result to lose precision when special hardware support is not provided.

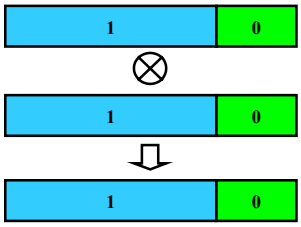


Figure 1. Fixed-point multiplication of 1 by 1

Figure 2. Multiplication results using integer multiplication

```
Input: a, b
Output: c

c = a * b;
sr(c, p);
Return (c);
```

Figure 3. Algorithm I

```
Input: a, b
Output: c

long long c;
c = a * b;
sr (c, p);
Return ((long) c);
```

Figure 4. Algorithm II

Therefore, to maintain full precision, another approach must be used.

Algorithm II, shown in Figure 4, appears to be another straightforward solution. However, implementing the 2*n*-bit (64-bit) multiplication operation on hardware that only supports 32-bit operations requires many assembly instructions. Table 1 in Section 4 shows that the instructions needed to perform the 64-bit multiplication include many integer multiplication instructions.

There is a similar problem for the division operation. Instead of the shift right operation, the shift left operation is needed for division after computation. An algorithm analogous to Figure 4 can be used for division. The assembly code generated shows that the straightforward algorithm consists of many integer multiplication and divisions.

Thus, for efficient computation, it is necessary to design efficient algorithms for the fixed-point arithmetic operations.

3. Our Algorithms

In this section, our algorithms for multiplication and division are presented. Let us denote bit_wise_or(a, b) as the bit-wise or operation of a and b. Our multiplication algorithm is shown in Figures 5 and 6. We consider two cases: (i) $p \le n/2$ and (ii) $p > n/2$.

```
Multiplication Algorithm (p ≤ n/2)
Input: a, b
Output: c

long ai, af, bi, bf;

ai = sr(a, p);
af = right p bits of a;
bi = sr(b, p);
bf = right p bits of b;

Return (sr(af * bf, p) + ai * b
          + af * bi);
```

Figure 5. Multiplication algorithm (p ≤ n/2)

```
Multiplication Algorithm (p > n/2)
Input: a, b
Output: c

long au, al, bu, bl;
au = sr(a, n/2);
al = right n/2 bits of a;
bu = sr(b, n/2);
bl = right n/2 bits of b;

Return (sr(al * bl, p)
  + sr(au * bl, p - n/2)
  + sr(al * bu, p - n/2)
  + sl(au * bu, n - p));
```

Figure 6. Multiplication algorithm (p>n/2)

At first glance, the proposed algorithm looks more complex than Algorithm II. However, even though Algorithm II contains only one multiplication and one shift operation in the C code, the computation must be implemented using many integer instructions because more precision than the processor directly supports must be handled. Thus, two registers are allocated for each data and many instructions are needed to perform the necessary operations. The code in our algorithm can be more

directly translated to assembly code, and, in Section 4, comparisons are shown.

Our algorithm for division is shown in Figure 7.

```
Division Algorithm
Input: a, b
Output: c

long ai, s, bu, bl;

ai = a/b;
s = a – ai * b;
bu = sr(a, 1);
bl = a bit_wise_and 1;

for i = 1 to  p
        t = sl(t, 1);
        a = a - bu;
        a = sl (a, 1);
        a = a - bl;

        if  (a < 0)
                a = a + b;
        else
                t = t bit_wise_or 1;
Return (bit_wise_or (s, t);
```

Figure 7. Division algorithm

In this algorithm, *ai* is the integer part and *s* is the fraction part of the result. The bit-wise or operation is performed to combine these two results. The algorithm requires one division, one multiplication, *p* additions, *p* subtractions, and *p* bit-wise logical operations. Thus, performance is better when *p* is small.

4. Implementation

In this section, the SLIIC Quick Look (QL) board, on which the experiments were performed, is briefly described. Then, implementation results of the algorithms are presented.

4.1. SLIIC QL Architecture

A block diagram of the SLIIC Quick Look (QL) board is shown in Figure 8. The board is implemented on a standard PCI form factor and contains eight Mitsubishi M32R/Ds, providing 640 MIPS of peak processing power and 16 MB of memory.

XA and XB are FPGAs that serve several purposes. First, they provide an interconnection network for the M32R/Ds. Second, XA and XB provide programmable logic that can be used for processor synchronization and performance measurement. Third, the FPGAs provide logic that facilitates communication with the host PC.

The SLIIC QL board communicates with the host PC using the SLIIC debugger. The SLIIC debugger, which runs on Windows NT, allows a user to start execution, read and write data to and from the PIMs, write data to the PIMs, and read counter data (there is a counter associated with each processor that is used for measurement of execution time), etc. More detailed information, such as supported commands and memory address mapping can be found in [13]. For application programmers, an application programming interface (API) is provided that includes a communication interface, barrier synchronization, and timer control.

4.2. Implementation Results

We implemented our fixed-point arithmetic algorithms on the SLIIC QL board, and the results are shown in this section. The gcc cross compiler was used with optimization level 2. For comparison purposes, Algorithm II (Figure 4) was also implemented. Table 1 shows the number of multiplication, division, and total instructions for each fixed-point operation. Note that most instructions except multiplication and division are executed in one cycle. Multiplication takes three cycles and division takes 36 cycles [9].

The results show that our algorithm takes significantly fewer instructions than Algorithm II. For the multiplication operation, the number of multiply instructions is reduced 33% and 50% and the number of total instructions is reduced 64% and 75% for p <= n/2 and p > n/2 respectively. The reduction is larger for the division operation where the number of divisions is reduced by 83% and the number of multiplications is reduced by 75%. The number of total instructions is reduced by 95%.

Table 1. Comparison of algorithms

Operation / # of Instructions	Multiplication			Division	
	Alg. II	Our		Alg. II	Our
		P≤ n/2	P> n/2		
Total	44	11	16	173	$4p+5$
Multiply	6	3	4	4	1
Division	0	0	0	6	1

The execution time for each operation is shown in Figure 9 and Figure 10. The result was obtained by averaging the execution times over 10 trials. The measurement was performed using hardware counters implemented in FPGAs. For division, the value of p was varied to measure the execution time for different data formats. The results show that our algorithms reduce execution times significantly. For multiplication, our algorithm takes only 18% to 30% of the execution time of the Algorithm II. For division, as expected from the analysis, the execution time for our algorithm is proportional to the value of p. However, in every case, our algorithm takes less time than the straightforward algorithm (only 15% to 76% of depending on p).

5. Conclusions

We have described efficient algorithms for arithmetic operations on a processor that does not support arbitrary fixed-point operations. The algorithms were analyzed and implemented on the SLIIC QL architecture, a single-board PIM-based multiprocessor with a programmable interconnect. The results show that the algorithms are much more efficient than straightforward approaches.

6. Acknowledgements

The authors gratefully acknowledge Steven Shank, Richard Chau, Walt Mazur, and Rick Pancoast of Lockheed Martin for providing the original CSLC code and fixed-point precision analysis.

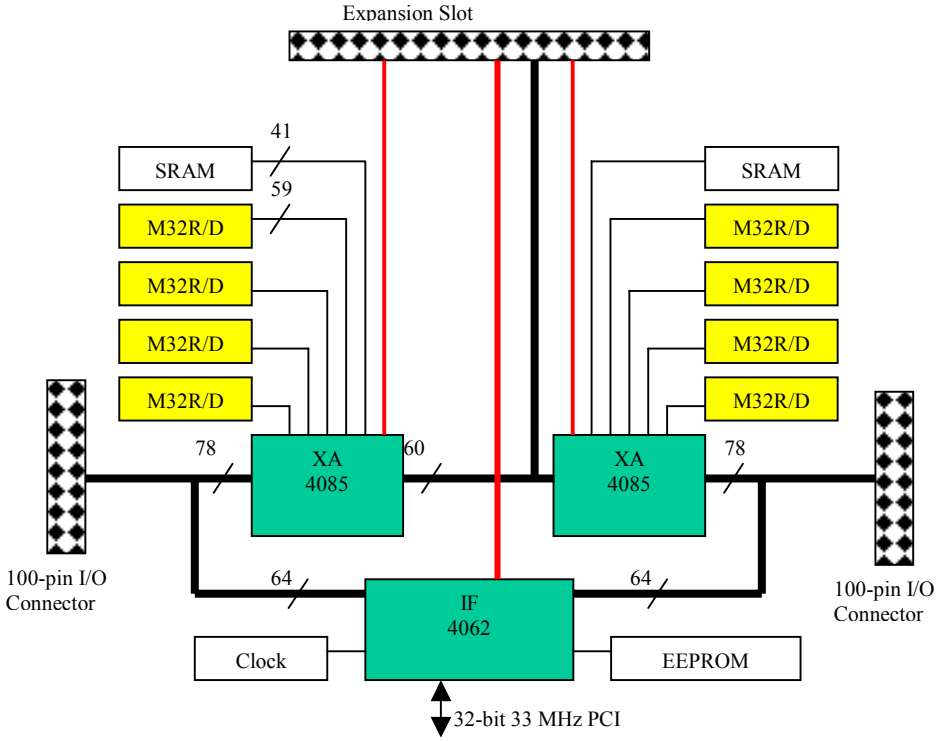


Figure 8. SLIIC architecture

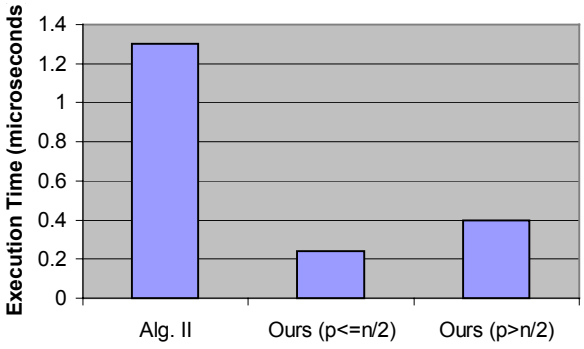


Figure 9. Multiplication

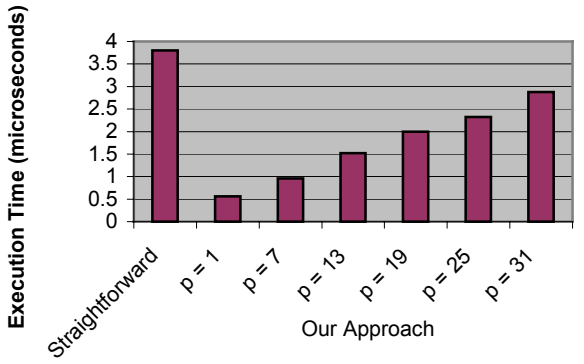


Figure 10. Division

7. References

[1] Advanced RISC Machines, "Fixed Point Arithmetic on the ARM," Application note 33, Document number: ARM DAI 0033A, September 1996.

[2] DARPA, Data Intensive Systems, http://www.darpa.mil/ito/research/dis/index.html, 2000.

[3] A. Gupta, J. L. Hennessy, K. Gharachorloo, T. Mowry, and W. D. Weber, "Computative Evaluation of Latency Reducing and Tolerating Techniques," Proc. 18th Annual International Symposium on Computer Architecture, Toronto, May 1991.

[4] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, J. Brockman, A. Srivastava, W. Athas, V. Freeh, J. Shin, and J. Park, "Mapping Irregular Applications to DIVA, a PIM-based Data-Intensive Architecture," Supercomputing '99, Portland, OR, November, 1999.

[5] J. Hennessy and D. A. Patterson, Computer Architecture: A Quantitative Approach, 2nd Edition, Morgan Kaufmann Publishers, Inc., 1996.

[6] C. Kozyrakis, "A Media-Enhanced Vector Architecture for Embedded Memory Systems," Technical Report #UCB/CSD-99-1059, UC Berkeley, July 1999.

[7] K. Mai, et al, "Smart Memories: A Modular Reconfigurable Architecture," ISCA 2000, Vancouver, BC, Canada, June 2000.

[8] Mitsubishi, M32000D4BFP-80 Data Book, http://www.mitsubishichips.com/data/datasheets/mcus/ mcupdf/ds/e32r80.pdf.

[9] Mitsubishi, Mitsubishi 32-Bit Single-Chip Microcomputer M32R Family Software Manual, July 1998.

[10] S. A. Przybylski, Cache and Memory Hierarchy Design: A Performance-Directed Approach, Morgan Kaufmann Publishers, San Mateo, CA, 1990.

[11] S. Rixner, W. J. Dally, U. J. Kapasi, B. Khailany, A. Lopez-Lagunas, P. R. Mattson, and J. D. Owens, "A Bandwidth-Efficient Architecture for Media Processing," 31st Annual International Symposium on Microarchitecture, Dallas, Texas, November 1998.

[12] A. J. Smith, "Cache Memories," Computing Surveys, Vol. 14, No. 3, pp. 473-530, 1982.

[13] J. Suh, S. P. Crago, C. Li, S. Shank, R. Chau, W. Mazur, and R. Pancoast, "SLIIC QL Technical Report," USC/ISI Technical Report, in preparation.

[14] J. Suh, C. Li, S. P. Crago, and R. Parker, "A PIM-Based Multiprocessor," IPDPS 2001, San Francisco, CA, 2001.

[15] J. Suh, M. Zhu, C. Li, S. P. Crago, S. F. Shank, R. H. Chau, W. J. Mazur, and R. Pancoast, "Implementations of Real-Time Data Intensive Applications on PIM-Based Multiprocessor Systems," Joint Workshop of EHPC and WPDRS, San Francisco, CA, 2001.

[16] Xilinx, http://www.xilinx.com/company/press/kits/pld/factsheet.htm, 2000.

PIM- and Stream Processor-based Processing for Radar Signal Applications

Jinwoo Suh and Stephen P. Crago
University of Southern California/Information Sciences Institute
3811 N. Fairfax Drive, Suite 200, Arlington, VA 22203
1-703-248-6160
{jsuh, crago}@isi.edu

ABSTRACT

The growing gap in performance between processor and memory speeds has created a problem for data-intensive applications. Recent approaches for solving this problem are processor-in-memory (PIM) technology and stream processor technology.

In this paper, we assess the performance of systems based on PIM and stream processors by implementing data-intensive applications. The implementation results are compared with the measured performance of conventional systems based on the PowerPC and Pentium processors. The results show that the performance of systems based on these processors is improved up to 70 compared with conventional systems for these data-intensive applications.

Categories and Subject Descriptors

C.4 [**PERFORMANCE OF SYSTEMS**]: Design studies, Performance attributes.

General Terms

Algorithms, Measurement, Performance, and Experimentation.

Keywords

Vector, Stream, corner turn, coherent side-lobe canceller, beam steering, radar signal processing, V-IRAM, and Imagine.

1. INTRODUCTION

Microprocessor performance has been doubling every 18-24 months, as predicted by Moore's Law, for many years [6]. This performance improvement has not been matched by DRAM (main memory) latencies, which have only improved by 7% per year [6]. This growing gap in performance between processor and memory speeds has created a problem for data-intensive applications.

To solve this problem, many methods have been proposed[1][3][9][10][11]. However, these methods provide limited performance improvement or even lower performance for some applications.

Processor-In-Memory (PIM) technology is a promising method for closing the gap between memory speed and processor speed for data intensive applications. PIM technology integrates a processor on a chip that uses DRAM technology. The integration of memory and processor on the same chip has the potential to decrease memory latency and increase the bandwidth between the processor and memory. PIM technology also has the potential to decrease other important system parameters such as power consumption, cost, and area. The V-IRAM chip [5] is a PIM research prototype being developed at the University of California at Berkeley. The V-IRAM contains one vector-processing unit and 8 Mbytes of DRAM in addition to a scalar-processing unit. There is a 512-bit data path between the processing units and DRAM. The target processor speed is 200 MHz, which will provide a peak performance of 3.2 GOPS (Giga Operations Per Second).

Another approach for handling the growing processor-memory gap is stream processing. In this approach, the data is routed through stream registers to hide memory latency, allow the re-ordering of DRAM accesses, and minimize the number of memory accesses. The Imagine chip [4] is a research prototype stream processor being developed at Stanford University. It contains eight clusters of arithmetic units that process data from a stream register file. The target processor speed is 500 MHz, which will provide a peak performance of 40 GOPS.

The V-IRAM and Imagine PCB boards are under development at ISI. The V-IRAM board contains one V-IRAM, and Imagine board contains two Imagine chips.

In this paper, we assess the performance of data-intensive radar processing applications on the V-IRAM and Imagine processors. We implemented the corner turn, beam steering, and coherent side-lobe canceller (CSLC) applications and measured the performance using cycle accurate simulators. We show that the speedup of the systems based on these processors is up to 70 compared with PowerPC and Pentium-based systems for these applications.

The rest of the paper is organized as follows. In Chapter 2, a PIM and a stream processor are briefly described. Also, the systems under development using these chips are presented. Chapter 3 describes three applications we implemented: the corner turn, coherent side-lobe canceller, and beam steerer. Also, the techniques that we used to improve the performance on these platforms are described. In Chapter 4, the implementation results are shown, and Chapter 5 concludes the paper.

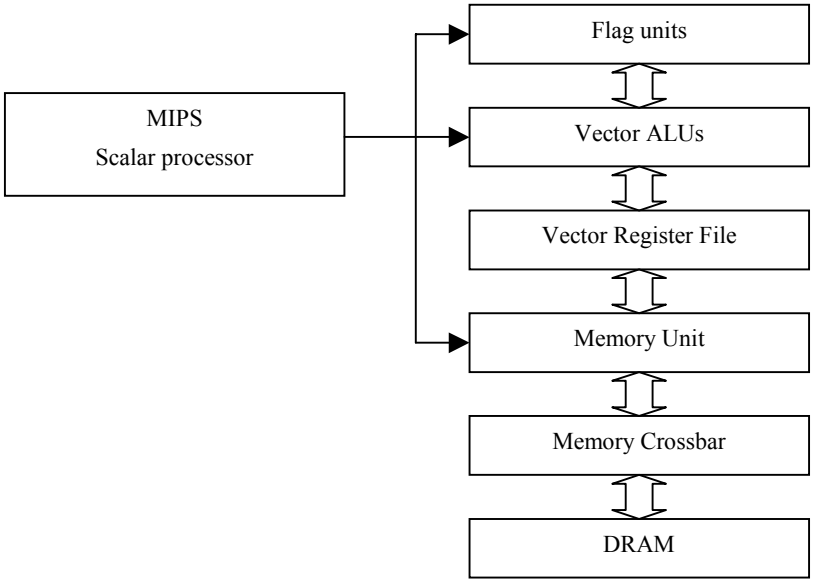


Figure 1. Simplified block diagram of V-IRAM

2. V-IRAM AND IMAGINE

2.1 V-IRAM

Processor-In-Memory (PIM) technology is a promising method for closing the gap between memory speed and processor speed. PIM technology integrates a processor on a DRAM memory chip, which uses DRAM technology. In conventional systems, the CPU and memory are implemented on different chips. Thus, the bandwidth between CPU and memory is limited since the data must be transferred through chip I/O pins and copper wires on a PCB. In a PIM-based system, the integration of memory and processor on the same chip has the potential to decrease memory latency and increases the bandwidth between the processor and memory. The power usage is smaller than traditional processor-memory chip pair since it takes less power to drive signals within a chip than between chips.

There are several PIM research prototypes being developed. One of them is the DIVA chip[4], which has 8 MB of DRAM and

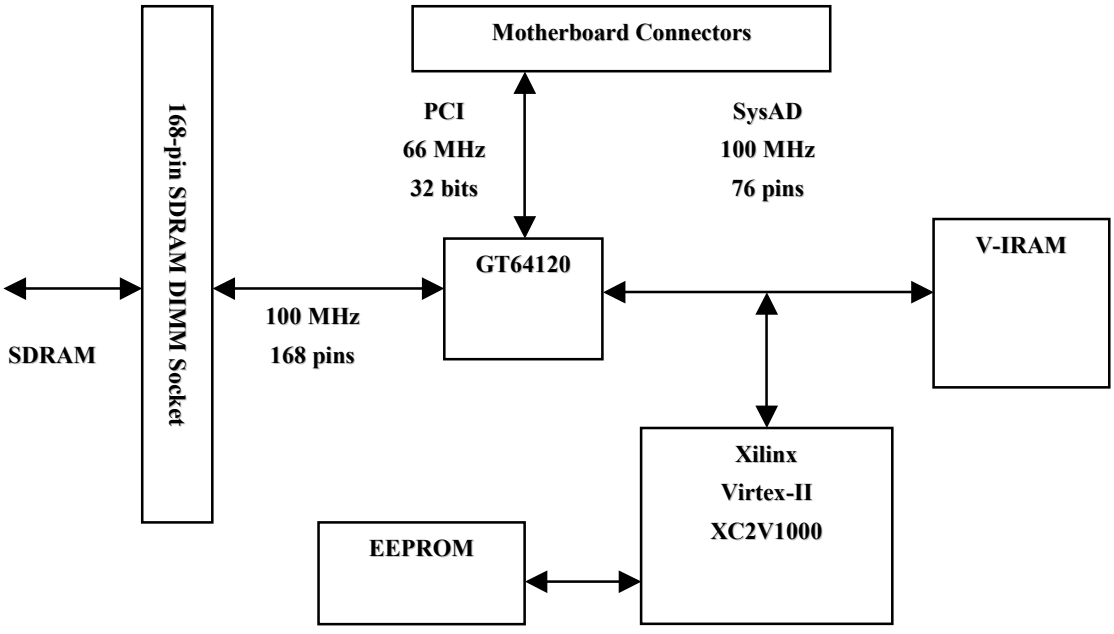


Figure 2. Block diagram of V-IRAM board

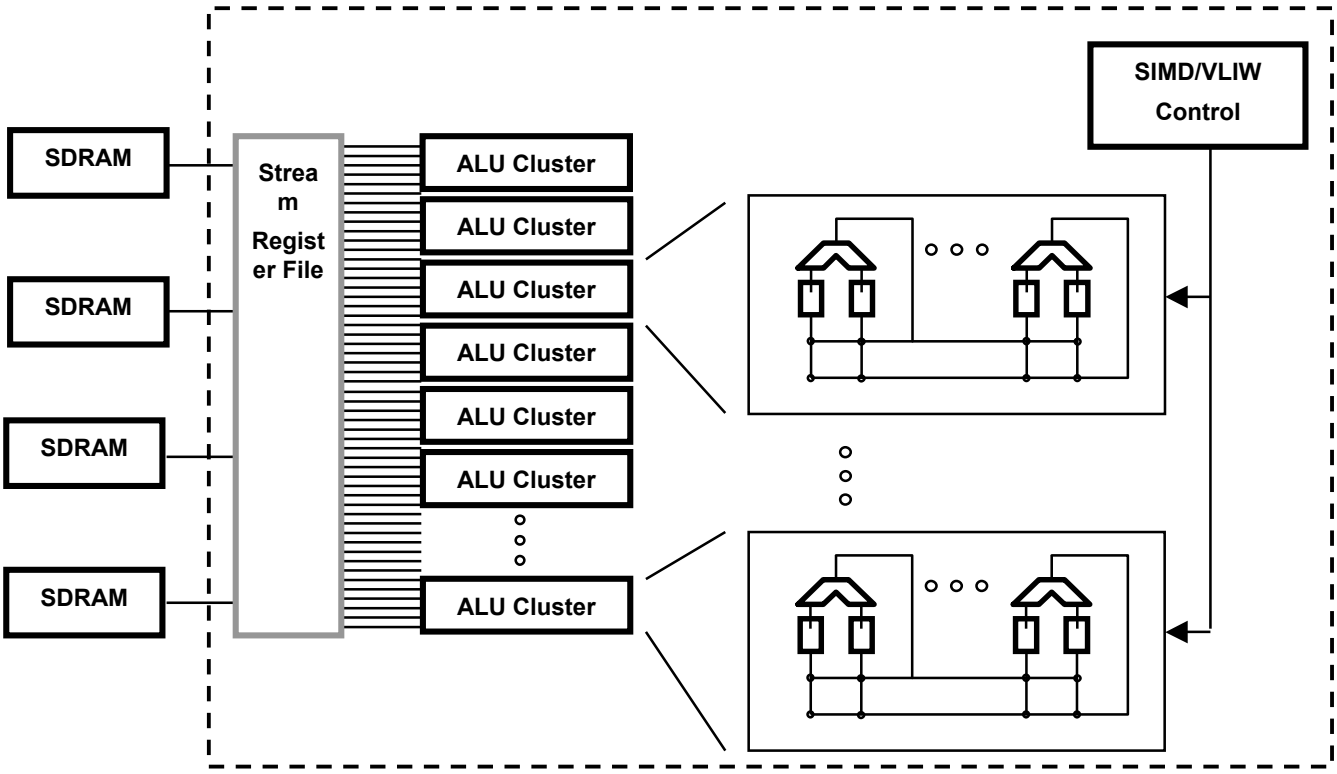


Figure 3. Block diagram of Imagine

1024 bits of total data path width.

A COTS PIM that is currently available is M32R/D [8]. A system using the M32R/D has been implemented at USC/ISI and the results have been reported [16]. In this system, eight M32R/D processors are interconnected using two FPGAs.

The IBM Blue Gene project is also investigating the use of PIMs in a high-performance parallel architecture[15]. The Blue Gene architecture is being developed for modeling the folding of human proteins. The Blue Gene project is developing multithreaded PIM processors.

The ADVISOR project investigated an algorithmic framework for the design of applications for PIM architectures [11]. The ADVISOR framework models the main characteristics of PIM chips and then, algorithms are designed based on the model. Example algorithms are reported for various applications.

The V-IRAM chip [6] is a research prototype PIM being developed at the University of California at Berkeley. The simplified architecture of the chip is shown in Figure 1. The V-IRAM contains one vector-processing unit and 8 Mbytes of DRAM in addition to a scalar-processing unit. The vector-processing unit contains two arithmetic units, two flag processing units, and two load/store units. These units are pipelined. Different kinds of operations have different number of stages. The functional units can be partitioned into several smaller units, depending on the arithmetic precision required. For example, a functional unit can be partitioned into 4 units for 64-bit operations or 8 units for 32-bit operations. There is a 512-bit data path between the processing units and DRAM. The DRAM is partitioned into two wings, each of which has four banks.

There is a crossbar switch between the DRAM and vector processor. The vector processor supports 91 instructions including arithmetic and vector processing. It also supports special vector instructions that help to obtain high performance on dot-product and FFT operations. The target processor speed is 200 MHz, which would provide a peak performance of 3.2 GOPS. The power consumption is expected to be about 2 W.

The block diagram of the V-IRAM board that is being developed at USC/ISI is shown in Figure 2. The board contains a V-IRAM PIM processor, one FPGA for glue logic, and one IF FPGA for the interface with a host computer.

2.2 Imagine

Imagine is a stream processing coprocessor that is being developed by Stanford University. The data is read in to the stream registers and sent to the cluster of arithmetic units where the data is processed. The processed data is stored back in the stream registers. The final data is stored in the memory. Figure 3 shows the block diagram of the Imagine.

The Imagine multiprocessor board is under development at ISI in collaboration with Stanford. The block diagram is shown in Figure 4. The board contains two Imagine chips, each of which is connected to local SDRAM. Each Imagine is connected to a PowerPC host processor through an FPGA chip. The Imagine chips are connected to a PowerPC host processor through an FPGA chip. The FPGA provides a means of connection between the PowerPC and Imagine by emulating an SDRAM interface on

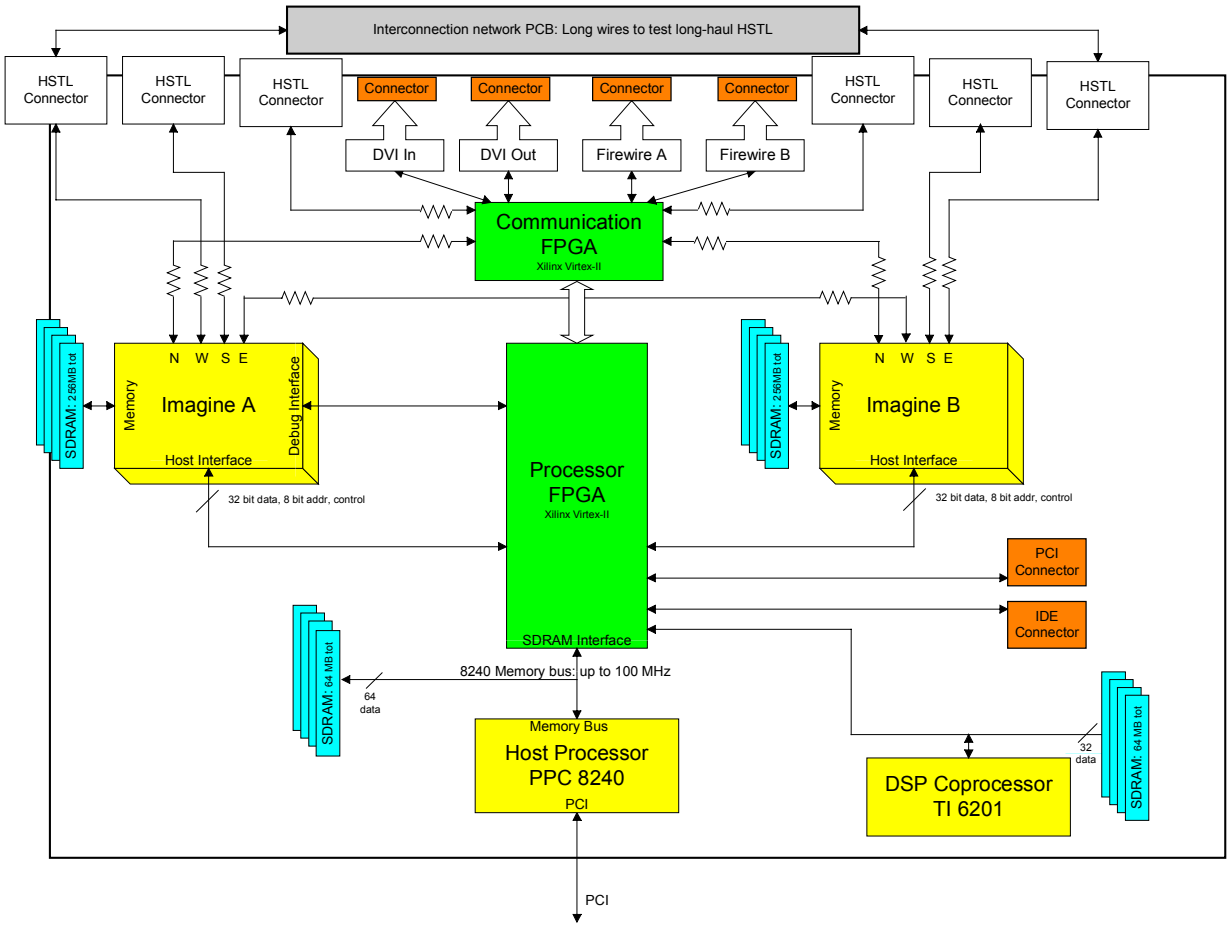


Figure 4. Imagine board

the PowerPC side and an SRAM interface on the Imagine side. It also provides a connection to the DSP chip. The PowerPC also has local SDRAM memory. The PowerPC processor communicates with a host PC through a PCI bus. Applications are compiled on the host PC and sent to the PowerPC through the PCI bus. The PowerPC performs address translation for the PCI memory space, so data written on the PCI memory space by the host PC is actually written on the local memory of the PowerPC. During execution of an application, when the PowerPC encounters kernel code that needs to be executed by an Imagine chip, the PowerPC sends instructions to the Imagine. The Imagine performs the computation and returns the data back to the PowerPC. The board also supports many I/O ports such as DVI, and those are connected to the Imagine chips through another FPGA.

3. Applications and Improvement Techniques

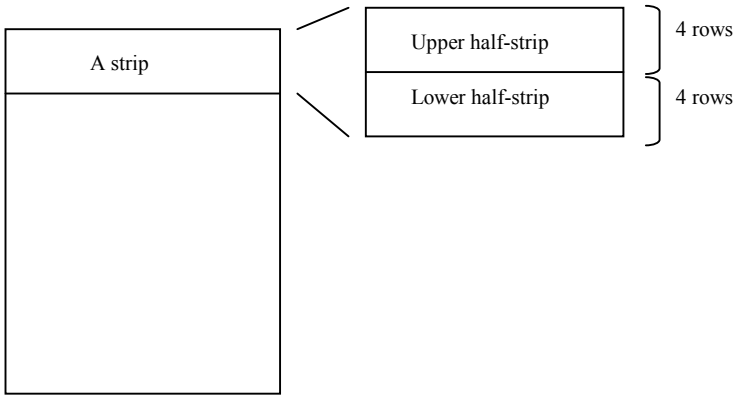
In this section, three benchmark applications are described. Also, the techniques used to improve the performance on V-IRAM and Imagine are presented. For comparison purposes, the C-version CSLC and beam steering codes that are being used for radar systems by Lockheed Martin are used as baselines.

3.1 Corner Turn

The corner turn is a matrix transpose operation. The data in the source matrix is transposed and stored in the destination matrix. The matrix size used for this paper, which was chosen to be larger than most caches, is 1024 x 1024 with 4-byte elements.

A straightforward implementation of the corner turn using two-level do-loops is simple, but the performance is degraded significantly due to the strided data accesses. In conventional processor systems, tiling is used to reduce the cost of load-store operations by re-ordering accesses to reduce working set size to better utilize a cache.

Since the term “column (row)” is used for both matrices and memories, we will distinguish them by denoting them as “matrix column (row)” and “memory column (row).”



(a) Strips and half-strips

0	1	2	3	4	5	…	N-1
N	N+1	N+2	N+3	N+4	N+5	…	2N-1
2N	2N+1	2N+2	2N+3	2N+4	2N+5	…	3N-1
3N	3N+1	3N+2	3N+3	3N+4	3N+5	…	4N-1

(b) Upper strip

0	N	2N	3N	1	N+1	2N+1	3N+1	2	N+2	2N+2	3N+2
4	N+4	2N+	3N+	5	N+5	2N+	3N+	6	N+6	2N+	3N+	…	4N-1

(c) Output stream from upper strip

4N	4N+1	4N+2	4N+3	4N+4	4N+5	…	5N-1
5N	5N+1	5N+2	5N+3	5N+4	5N+5	…	6N-1
6N	6N+1	6N+2	6N+3	6N+4	6N+5	…	7N-1
7N	7N+1	7N+2	7N+3	7N+4	7N+5	…	8N-1

(d) Lower strip

4N	5N	6N	7N	4N+1	5N+1	6N+1	7N+1	4N+2	5N+2	6N+2	7N+2
4N+	5N+	6N+	7N+	4N+	5N+	6N+	7N+	4N+	5N+	6N+	7N+	…	8N-1

(e) Output stream from lower strip

0	N	2N	3N	4N	5N	6N	7N	1	N+1	2N+1	3N+1
4N+1	5N+1	6N+1	7N+1	2	N+2	2N+2	3N+2	…	8N-1

(f) Combined strip

Figure 5. Corner turn on Imagine

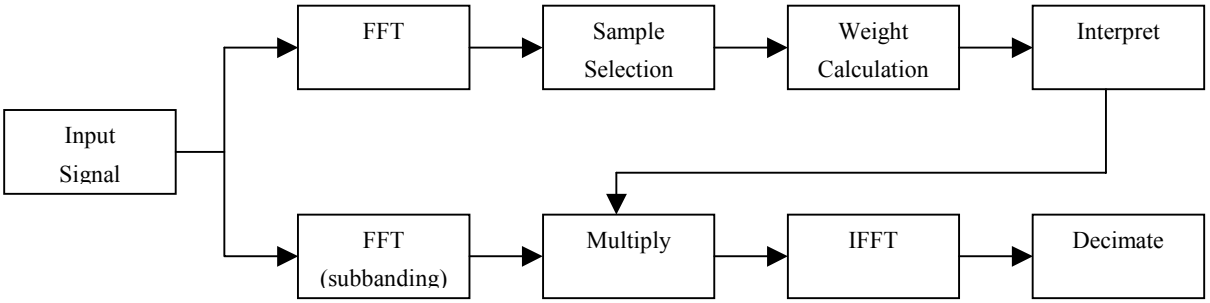


Figure 6. Coherent side-lobe canceller

The tile size we chose for our corner turn implementation on V-IRAM is 16 x 16 element matrix. The selection of tile size depends on the number of vector registers and the memory configuration. In the load operations of our implementation, each column in the tile is loaded into each register in strided mode. Then, the data in the registers are stored as a row in sequential mode. Even though we use the strided loads, the effective performance is as good as the sequential access because of the method described below.

When the first column in the tile is loaded, the load operation for each data stalls a few cycles while the memory column is accessed. However, when the second column in the tile is accessed, if the memory columns accessed previously are not pre-charged, then, the second matrix column can be accessed in one cycle. This is true for the third and all of the remaining matrix columns in the tile. The V-IRAM provides up to eight open columns. Thus, it is possible to keep the columns open when all of the memory columns accessed are in different memory row/wing combinations.

Thus, to limit the number of open columns to eight, we first partition the tile into two half-tiles: upper and lower. All data in the upper half-tile is read into the registers before the data in the lower half-tile. Since the size of the upper half-tile is 8 x 16, it is possible to keep all columns open. Also, we need to ensure that the wing-matrix combination does not appear more than once for the half-tile; otherwise, the previously opened column must be pre-charged and performance is degraded significantly. Thus, we used matrix padding to place data in different memory rows and wings. By using this algorithm, the performance of the stride access can be as fast as sequential access.

On the Imagine processor, we use the following technique to leverage the streaming capabilities. We partition the matrix into strips of data. Each strip consists of eight rows of data. For each strip, we read the data in the strip and do a transpose. Since the data in the source matrix is 8 x N elements, where N is the number of columns in the matrix, the transposed data is N x 8. The transposed data is stored in the destination matrix. This is explained in more detail in the following paragraphs.

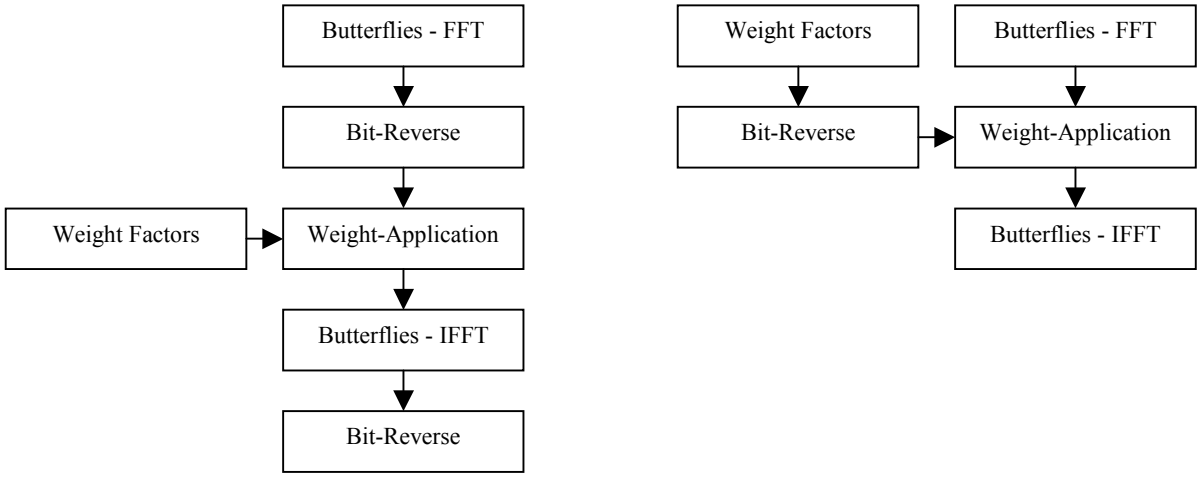
The strip is conceptually partitioned into two half-strips: an upper half and a lower half (see Figure 5). We first perform the corner turn for the upper half-strip ((b) and (c)). We read the four matrix rows and do the transpose using communication units in the clusters. For this operation, four input streams and one output stream are used. Since the rows are read sequentially, there is no performance degradation. The same operation is performed for the lower half-strip ((d) and (e)). Then, the two output streams are read and permuted using the communication unit in the clusters (f). The strip is written into the destination matrix. During the write operation, the unit of data is eight elements and the stride of data accesses is N. When each row in the strip is written, the data is sequentially stored, thus, we can obtain the maximum possible bandwidth. The cycles lost due to the stride mode for the write operation is inevitable since it is a characteristic of DRAM that the pre-charge time is required whenever memory rows are accessed.

On Imagine, it is not possible to perform a corner turn by reading eight input streams simultaneously since the current Imagine implementation limits the total number of streams to eight; if eight rows are read, no streams are left for an output stream.

3.2 Coherent Side-Lobe Canceller (CSLC)

CSLC is a radar signal processing application used to cancel jammer signals caused by one or more jammers. To cancel jammer signals that appear as side-lobes in the frequency domain, one auxiliary channel is needed per jammer signal.

The block diagram of the signal processing is shown in Figure 6. The operations in the upper half of the figure are known as weight calculations and the operations in the lower half are weight applications. To cancel the side-lobe, the weight factor is calculated using the signal from the auxiliary channel. Then, the main signal is partitioned into several sub-bands in the time domain. Each sub-band is then converted to the frequency domain using the FFT (sub-banding). Weight factors are multiplied with the output of the FFT operation to cancel the side-lobe. An inverse FFT is later performed on the output data. Most of the computation time is spent on the FFT and IFFT operations. In our implementation, only the weight application is implemented.



(a) Conventional

(b) Bit-reversed weight factors

Figure 7. New CSLC implementation

The following parameters are used for the implementation: four input channels, two main channels, and two auxiliary channels. Each channel has 8 K data samples. All computations are done using floating-point precision. The data is partitioned into 73 overlapped sub-bands, each of which contains 128 samples. For sub-banding, a 128-sample FFT is used.

To improve CSLC performance, we used several techniques: a combination of radix-4 and radix-2 FFT, hand optimization of assembly code for the FFT operation, reducing the number of bit-reverse operations, and eliminating load-store operations between computational stages.

Since the majority of computation time on the CSLC is spent on the FFT operation, we improved the performance of the FFT by using the appropriate FFT algorithms for each architecture.

On V-IRAM, a radix-4 FFT is used. Note that since the size of the FFT for the CSLC is 128, which is not power of 4, we used three stages of radix-4 FFT and one stage of radix-2 FFT. Since the current version of the V-IRAM compiler does not vectorize the FFT code written in C optimally, we hand-assembled the FFT to obtain the maximum performance using vector instructions. For example, there are instructions that are suitable for the FFT butterfly that the current compiler does not use for the FFT compilation, such as vhalfup, which shuffles data between two vector registers.

On the Imagine, as for the V-IRAM, a combination of the radix-4 FFT and the radix-2 FFT is used. We wrote two kernels for radix-4 and radix-2 FFT. We used the FFT algorithm that has the input and output pattern that is most suitable for stream processors. After each butterfly operation, the data is exchanged among clusters using a cluster communicator to arrange data appropriately. In addition to the optimization of the FFT itself, we also removed the bit-reverse operations. Instead of bit-reversing the result of the FFT, the weight factors are bit-reversed in the weight application processing. This is shown in Figure 7.

The number of bit-operations for the straightforward method is 2N, where N is the number of data sets on which the FFT, weight application, and IFFT are performed (N=73 in our implementation). However, in the new algorithm, the number of bit-reverse operations is only one. Therefore, the bit-reverse operation cost is reduced by a factor of 146. This also enables us to eliminate the load-store operations between the FFT, weight application, and IFFT.

3.3 Beam Steering

Beam steering is a radar processing application that directs a phased-array radar in an arbitrary direction without physically rotating the antenna. Figure 8 shows a one-dimensional beam steering operation. A real system consists of a two dimensional array of antenna elements populated on a plane. In the system, many small antenna elements transmit the signal with different phases. In the figure, each of the three antenna elements transmits a signal with phase shift of d * sin θ between adjacent elements. By choosing phases, the antenna direction can be controlled. The

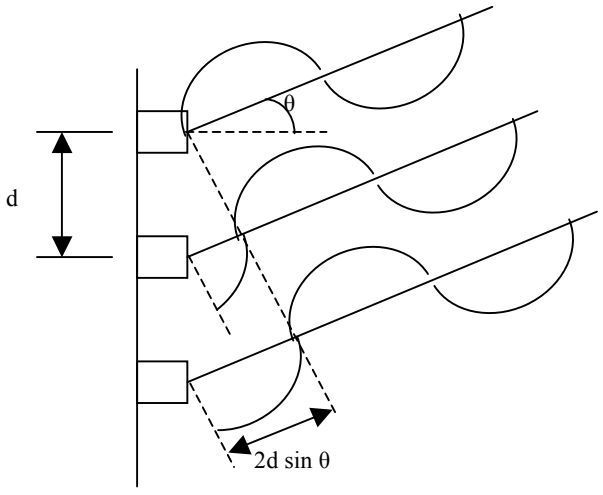


Figure 8. Beam steering

computation of the phase for each antenna element involves many load, store and arithmetic operations.

In our implementation, the following parameters are used. The number of antenna elements is 1608. Each element can direct the signal up to 4 directions per dwell where a dwell is a period. The phase needs to be calculated for each direction. Depending on the signal frequency and temperature, calibration data needs to be incorporated in the calculation of the phases. In our implementation, four calibration bands are processed.

As for other applications, we used hand-vectorization of the main portion of the beamsteering on V-IRAM. Note that the current compiler is still a prototype and it may be able to vectorize these in the future. For the Imagine, a kernel is written that utilize the clusters

4. EXPERIMENTAL RESULTS AND ANALYSIS

In this section, the implementation results of the corner turn, CSLC, and beam steering applications are presented. Performance of these applications is estimated using cycle-accurate simulators provided by the V-IRAM and Imagine teams. For comparison purposes, actual measurements of the application performance were taken using a single node of a PowerPC-based multiprocessor system and a Pentium III system. Applicable performance improvement techniques were also applied to these platforms.

The PowerPC results are obtained using a PowerPC-based multiprocessor, the CSPI 2741 [2]. Each CSPI 2741 consists of two boards. Each board contains two PowerPC 750 (400 MHz) processors and a LaNAI network interface. The processors are interconnected through a Myrinet network. The gcc compiler is used for compilatopm.

The Pentium results are obtained using a PC running the Linux operating system. The CPU in the system is Pentium III running at 733 MHz. The gcc compiler is used for compilation.

Table 1. Experimental results

	Corner Turn (MB/sec)	CSLC (msec)	Beam Steering (msec)
PPC G3 (400 MHz)	21.0	16.6	3.76
Pentium III (733 MHz)	83.9	32.2	4.74
V-IRAM (200 MHz)	1441.8	2.57	0.31
Imagine (500 MHz)	1199.1	0.77	0.30

In Table 1, the implementation results of corner turn, beam steering, and CSLC are shown. The speedup is shown in Figure 9. Figure 9 shows that V-IRAM and Imagine provide speedups up to 70 compared with a PowerPC system even though their clock frequencies are not particularly fast. The results show that the V-IRAM performs better than the Imagine on a corner turn. This is because the V-IRAM has higher bandwidth between memory (which is on-chip on the V-IRAM and off-chip on the Imagine) and the processing unit. However, the Imagine has higher computational performance, which is reflected in the performance of the CSLC, which is more computation-intensive. The V-IRAM and Imagine have similar performance on the beam-steering application because of the balance between memory lookups and computation of that application.

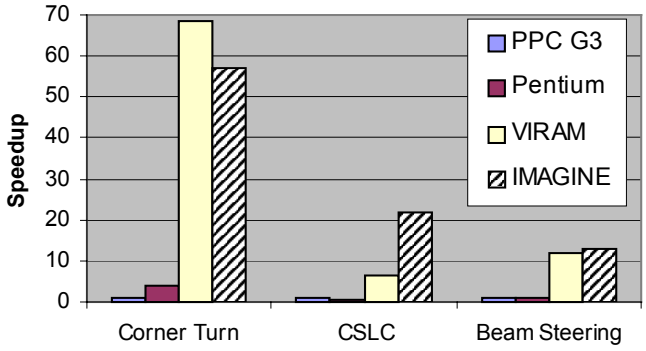


Figure 9. Speedup

In a real system, the current implementation of the V-IRAM may take less space than the current implementation of the Imagine since it has a scalar processor and internal DRAM on-chip and does not need external memory if the application fits in the memory.

5. CONCLUSION

We have presented simulated performance results for data-intensive radar processing applications on systems based on the V-IRAM PIM and the Imagine stream processor and compared them to conventional systems. The results show the potential advantages of the new technologies on data intensive applications.

We have presented the implementation results of the real-time data intensive applications, coherent side-lobe canceller and beam steering, on both innovative data-intensive systems (V-IRAM and Imagine) and conventional systems (PowerPC and Pentium III). The implementation results show the speedup using these chips provide up to almost 70 compared with the PPC-based system.

6. ACKNOWLEDGMENTS

The authors gratefully acknowledge the UC Berkeley IRAM team and the Stanford Imagine team for the use of their compilers and simulators and their generous help. The authors specifically acknowledge Brian Patrick Towles for providing initial idea of the corner turn on Imagine. The authors also acknowledge Rick Pancoast, Steve Shank, Walt Mazur, and Joe Racosky of Lockheed Martin NE & SS for providing the applications.

Effort sponsored by Defense Advanced Research Projects Agency (DARPA) through the Air Force Research Laboratory, USAF, under agreement number F30602-99-1-0521. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not interpreted as necessarily representing the official policies or endorsement, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), Air Force Research Laboratory, or the U.S. Government.

7. REFERENCES

[1] C. Conti, D. H. Gibson, and S. H. Pitowsky, "Structural aspects of the System/360 Model 85, Part I: General Organization," IBM Systems Journal, Vol. 7, No. 1, pp. 2-14, 1968.

[2] CSP Inc., "2000 Series Hardware Manual S2000-HARD-001-01," CSP Inc., 1999.

[3] A. Gupta, J. L. Hennessy, K. Gharachorloo, T. Mowry, and W. D. Weber, "Computative Evaluation of Latency Reducing and Tolerating Techniques," Proc. 18th Annual International Symposium on Computer Architecture, Toronto, May 1991.

[4] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, J. Brockman, A. Srivastava, W. Athas, V. Freeh, J. Shin, and J. Park, "Mapping Irregular Applications to DIVA, a PIM-based Data-Intensive Architecture," Supercomputing '99, Portland, OR, November 1999.

[5] B. Khailany, et. al., "Imagine: Signal and Image Processing Using Streams," HOT Chips 12, Stanford, CA, August 2000.

[6] C. Kozyrakis, "A Media-Enhanced Vector Architecture for Embedded Memory Systems," Technical Report # UCB/CSD-99-1059, UC Berkeley, July 1999.

[7] J. Hennessy and D. A. Patterson, Computer Architecture: A Quantitative Approach, 2nd Edition, Morgan Kaufmann Publishers, Inc., 1996.

[8] Mitsubishi Microcomputers, M32000D4BFP-80 Data Book, http://www.mitsubishichips.com/data/datasheets /mcus/ mcupdf/ds/e32r80.pdf.

[9] Motorola, EC603e Embedded RISC Microprocessor Hardware Specifications, http://ebus.mot-sps.com/brdata/

[10] D. A. Patterson, J. L. Hennessy, Computer organization and design: the hardware/software interface, Morgan Kaufmann, 1994.

[11] M. Penner and Viktor K. Prasanna, "Cache Friendly Implementations of Transitive Closure", In Proc. of International Conference on Parallel Architectures and Compilation Techniques, September 2001.

[12] S. A. Przybylski, Cache and Memory Hierarchy Design: A Performance-Directed Approach, Morgan Kaufmann Publishers, San Mateo, CA, 1990.

[13] S. Rixner, W. J. Dally, U. J. Kapasi, B. Khailany, A. Lopez-Lagunas, P. R. Mattson, and J. D. Owens, "A Bandwidth-Efficient Architecture for Media Processing," 31st Annual International Symposium on Microarchitecture, Dallas, Texas, November 1998.

[14] A. J. Smith, "Cache Memories," Computing Surveys, Vol. 14, No. 3, pp. 473-530, 1982.

[15] M. Snir, "Blue Gene System Overview," Fourth Annual High Performance Embedded Computing Workshop," Boston, MA, September 2000.

[16] J. Suh, S. P. Crago, C. Li, and R. Parker, "A PIM-based Multiprocessor System," International Parallel and Distributed Processing Symposium, San Francisco, CA, 2000.

[17] Xilinx, http://www.xilinx.com/company/press/kits/pld/ factsheet.htm, 2001.

An Efficient Algorithm for Out-of-Core Matrix Transposition

Jinwoo Suh, *Member*, *IEEE*, and Viktor K. Prasanna, *Fellow*, *IEEE*

Abstract—Efficient transposition of Out-of-core matrices has been widely studied. These efforts have focused on reducing the number of I/O operations. However, in state-of-the-art architectures, memory-memory data transfer time and index computation time are also significant components of the overall time. In this paper, we propose an algorithm that considers the index computation time and the I/O time and reduces the overall execution time. Our algorithm reduces the total execution time by reducing the number of I/O operations and eliminating the index computation. In doing so, two techniques are employed: writing the data onto disk in predefined patterns and balancing the number of disk read and write operations. The index computation time, which is an expensive operation involving two divisions and a multiplication, is eliminated by partitioning the memory into read and write buffers. The expensive in-processor permutation is replaced by data collection from the read buffer to the write buffer. Even though this partitioning may increase the number of I/O operations for some cases, it results in an overall reduction in the execution time due to the elimination of the expensive index computation. Our algorithm is analyzed using the well-known Linear Model and the Parallel Disk Model. The experimental results on Sun Enterprise, SGI R12000, and Pentium III show that our algorithm reduces the overall execution time by up to 50 percent compared with the best known algorithms in the literature.

Index Terms—Matrix transpose, data transfer time, index computation time, I/O time, out-of-core, execution time.

1 Introduction

Matrices are typically stored in row-major order. To access all the elements in a column of a matrix, a straightforward approach will access the disk M times, where M is the number of rows of the matrix. One approach to avoid this is to transpose the matrix and store the transposed matrix on the disk. The problem is of constructing a specific permutation of a given sequence when only a part of the sequence can be kept in the main storage and operated on at the same time, while minimizing the number of I/O operations [13]. Hence, matrix transpose is a key primitive in a wide variety of scientific computations. Matrix transpose is also a fundamental operation in adaptive signal processing [3], [9], [16], [21], [22]. In such applications, the typical data size that is stored on the disk is of the order of TeraBytes. To store such data, high-performance computing platforms employ RAID [5] disk systems.

Two models have been widely used in the literature to abstract the behavior of disk systems: the Parallel Disk Model (PDM) [28] and the Linear Model (LM) [19]. The PDM is well-suited to model I/O systems such as the RAID [5]. In PDM, the data access cost is represented as $\lceil m/(DB) \rceil \times T_b$, where m is the data size, D is the number of disks, and T_b is time to transfer a block of data (B) between memory and disk. In the Linear Model, the cost is represented as $T_s + m\tau$, where T_s is the startup time, m is the data size, and τ is the data transfer time per unit data.

In this paper, we propose an efficient algorithm for transposing large-scale matrices (out-of-core matrix transpose). A matrix of size $N \times N$ initially resides on the disk. The matrix is to be transposed and stored in another array. The size of the available main memory, M, is smaller than the matrix size. Without loss of generality, we assume that the matrices are stored in row-major order. Several researchers have studied the out-of-core matrix transpose problem. A straightforward algorithm performs matrix transpose using $O(N^{3/2})$ I/O operations when $M = O(N)$. Eklundh [13] proposed an algorithm that has $O(N \log N)$ I/O complexity assuming $B = N$. Ari [2] modified the algorithm in [14] to reduce the number of I/O operations at the expense of increased number of stages (passes). Floyd [14] has derived upper and lower bounds on the number of I/O operations when $M = 2B$. Aggarwal and Vitter [1] have shown a lower bound on the number of I/O operations for the general case using PDM. Vitter and Shriver [28] proposed PDM and also provided an asymptotic lower bound for several algorithms including permutation, of which matrix transpose is a special case. Cormen et al. [8] showed asymptotically equal lower and upper bounds for the number of I/O operation for bit-matrix-multiply/complement (BMMC) permutations. Kaushik et al. [19] reduced the number of I/O operations by 25 percent compared with the algorithm in [14] by combining two read operations.

All these efforts focus on reducing the number of I/O operations only. However, the main costs in state-of-the-art architectures consist of not only the time for I/O, but also the memory-memory data transfer time and the index computation time.

- *J. Suh is with the USC Information Sciences Institute, 3811 N. Fairfax Dr., Suite 770, Arlington, VA 22203. E-mail: jsuh@isi.edu.*
- *V.K. Prasanna is with the Department of Electrical Engineering-Systems, EEB-200C, University of Southern California, Los Angeles, CA 90089-2562. E-mail: prasanna@usc.edu.*

Manuscript received 11 Dec. 2000; revised 9 July 2001; accepted 29 Aug. 2001.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 113280.

Our out-of-core matrix transpose algorithm reduces the total execution time by reducing both the number of I/O operations and the index computation time. The reduction in the number of I/O operations is achieved by using efficient data read and write schedules and by balancing the number of read and write operations. We analyze the complexity of our algorithm using the well-known Parallel Disk Model (PDM) and the Linear Model (LM).

To eliminate the index computation cost, our algorithm partitions the available memory into two buffers (read and write buffers). The expensive in-processor permutation is replaced by data collect operations. The write operations and collect operations are scheduled efficiently to reduce the overall time. The size of each buffer is determined by the available memory size and the factorization of N. By using these techniques, the index computation is replaced by inexpensive do-loops (see Section 5.2).

We implemented the algorithm on an SGI R12000, a Sun Enterprise based on UltraSPARC-III, and a Pentium III-based platform at the University of Southern California. The experiments were carried out for available main memory sizes ranging from 16 MB to 64 MB and data sizes ranging from 128 MB to 8 GB. The results show that our algorithm reduces the overall execution time by up to 50 percent.

The organization of the remainder of this paper is as follows: In Section 2, two well-known disk models are briefly described. In Section 3, previous algorithms for large scale matrix transposition are discussed. Overview of our algorithm is presented in Section 4. Our algorithm is described in detail in Section 5. Experimental results, as well as comparisons with previous algorithms, are presented in Section 6. Section 7 discusses a further extension of our algorithm and Section 8 concludes the paper.

2 Disk Models

State-of-the-art disk systems employ sophisticated hardware and perform several optimizations to reduce the I/O time. For example, many of these systems employ a disk buffer, a library buffer, and a controller and perform access reordering. Each of the above system features needs several parameters to describe its behavior and such a model will be too complex to be useful.

Two models of disk systems that capture the key characteristics of such systems have been widely used in the literature. One of them is the Parallel Disk Model (PDM) [28]. It models the low-level behavior of disk systems using few parameters: block size (B), which is the size of data transferred between disk and memory in one I/O operation, number of disks (D), the number of processors (P), the size of memory (M), and the amount of data transferred (m). In this paper, P is assumed to be one. The total time for data transfer between disk and memory can be represented as $\lceil m/(DB) \rceil \times T_b$, where T_b is the time to transfer a block of data between memory and disk.

In another model [19], two costs are considered: startup time and data transfer time. The startup time is a fixed time for setting up the data transfer between memory and disk. The rest of the cost is proportional to the amount of data transferred. Thus, the access cost can be represented as $T_s + m\tau$, where T_s is the startup time, m is the data size, and τ is the time to transfer unit data. Typically, T_s is in the order of msec and τ is tens of nsec/byte.

3 Previous Algorithms

In this section, for the sake of completeness, two well-known algorithms are briefly described. These two algorithms provide the best performance over many other algorithms. The algorithm in [1] has been designed using the PDM and the algorithm in [19] has been designed using the LM. In Section 4, our algorithm is compared with these algorithms.

3.1 Matrix Transpose

In the matrix transpose problem, an input matrix of size $N \times N$ initially resides on the disk, where $N = \prod_{s=0}^{t-1} r_s$, for some $t > 0$, where r_s is a positive integer. If N is a prime number, we can add dummy rows to make N to be nonprime. The input matrix is to be transposed and stored in another array. M, the size of the available memory, is smaller than the input matrix size. Throughout this paper, to illustrate the key ideas, we use square matrices. However, the algorithms can be easily extended to rectangular matrices as well, using the technique in [6], [19]. Also, for the sake of simplicity, throughout the paper, we assume that the result of all arithmetic operations are integers.

3.2 Aggarwal and Vitter's Algorithm

Aggarwal and Vitter [1] showed a lower bound on the number of I/O operations to perform matrix transpose. In this algorithm, as many blocks as can fit in the available memory are read into memory. Then, the data are permuted and written onto the disk. The number of I/O operations for this approach is shown in Table 3. The number of I/O operations is $\frac{2N^2}{B} \lg_{\frac{M}{B}} \frac{N^2}{B}$. The asymptotic complexity of the algorithm is $N^2 \lg(N^2)$, when M and B are constants. Note that the asymptotic complexity of the algorithm is analogous to $N \lg N$ bound for sorting on the RAM model [7], [26], where N is the number of data elements.

In this algorithm, r_s is restricted to be $\leq M/B, 0 \leq s < t$. This is because, if $r_s > M/B$, a block must be stored $\lceil \frac{r_s}{M/B} \rceil$ times per iteration instead of storing it once. This results in a considerable increase in the number of I/O operations. In our algorithm, we relax this restriction by developing a technique to use a larger block size. Also, Aggarwal's algorithm does not minimize the index computation time. Index computation is needed to perform permutation of the data in memory. The pseudocode for Aggarwal and Vitter's algorithm is given in Fig. 1.

3.3 Kaushik et al.'s Algorithm

In this algorithm [19], there are t stages, where $N = \prod_{s=0}^{t-1} r_s$. An example is shown in Fig. 3. The number in each small square denotes a data element. The arrow indicates read and write operations.

Each stage consists of N^2/M steps. In each step, M/N rows are read into memory and permutation of the data is performed in the memory. In stage s, $0 \leq s < t$, the

```
1  for s = 0 to lg_{M/B} min(B, N^2/B) - 1
2      for j = 0 to N^2/M - 1
3          Read M/B blocks;
4          Permute data in memory;
5          Write M/B blocks;
```

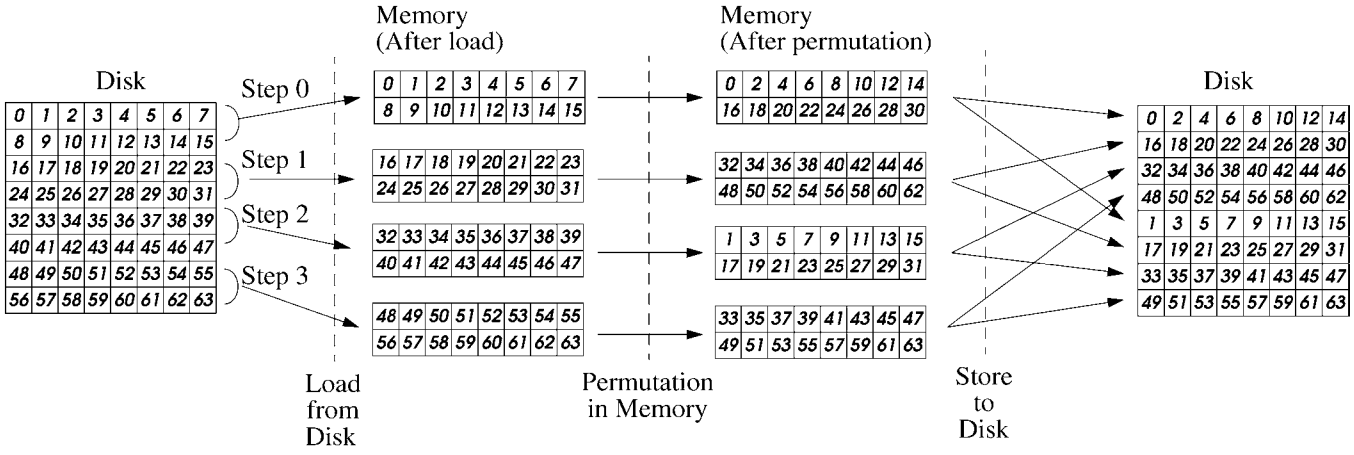
Fig. 1. Pseudocode for Aggarwal and Vitter's algorithm

```
1  for s = 0 to t-1
2      for j = 0 to N^2/M - 1
3          Read M amount of data;
4          Permute data in memory;
5          for k = 0 to r_s - 1
6              Write M/r_s amount of data;
```

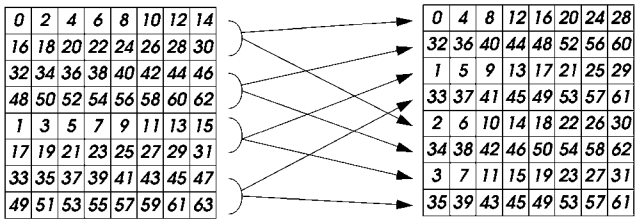
Fig. 2. Pseudocode for Kaushik et al.'s algorithm (LM).

data are written back onto the disk in r_s write operations. Thus, the number of read (write) operations in each step is 1 (r_s). The total number of I/O operations using the LM and the PDM are shown in Table 3.

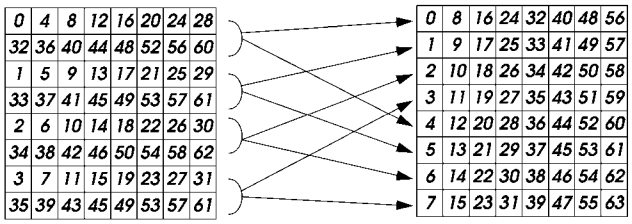
Although the number of I/O operations and the time to transfer data between memory and disk are considered, the total number of read and write operations are not optimized. Also, the index computation time is not minimized. The pseudocode for Kaushik et al.'s algorithm is given in Fig. 2.



(a)



(b)



(c)

Fig. 3. An illustrative example ($N = 8 = \prod_{s=0}^{2} 2$ and $M = 16$)—Kaushik et al.'s algorithm.

```
1   for s = 0 to t-1 step = 1// for each stage
2       for u = 0 to N^2/M-1 step = 1// for each step
3           Read M units of data from disk using read schedule RS(s, u);
4           Permute the data in memory;
5           Write M units of data to disk using write schedule WS(s, u);
```

Fig. 4. Overview of the Algorithm.

4 Overview of our Algorithm

We present an overview of our approach in this section. Section 5 provides the details of our approach and analysis using PDM and LM.

One of the key features of our algorithm is the reduction in the total number of I/O operations, which is achieved by means of efficient read and write schedules and corresponding data permutation in the memory. For example, in [19], there are three I/O operations (one read operation and two write operations) in each step when $M = 2N$ and $B = N$. Our algorithm requires only a single write operation in each step compared with two write operations as in the case of the algorithms in [1], [19]. The concept of a step is explained in detail in Section 5. Since our algorithm consists of the same number of steps as in the previous algorithms, there is a considerable reduction in the total number of write operations.

Another technique used in our algorithm is the balancing of the numbers of read and write operations. In balancing the numbers of read and write operations, the key idea is that the total number of I/O operations can be reduced by reducing the number of write operations at the expense of an increased number of read operations. For example, when $r_s = 32$, in each step, the number of read (write) operations in [19] is 1 (32). In our algorithm, we increase the number of read operations to nine in order to reduce the number of write operations to nine. This results in a 45 percent reduction in the total number of I/O operations. Note that a straightforward method to balance the numbers of read and write operations reduces the total number of I/O operations by only one (see Section 5). The data that are written onto the disk in z_s write operations in the previous algorithm is written in one write operation in our algorithm. Thus, there is a reduction in the number of write operations by a factor of z_s, where z_s is an integer ≥ 2. In a subsequent read operation (that is, in the next stage), the data is read in z_s read operations. By choosing an optimal value of z_s, the total number of I/O operations is reduced.

In the previous algorithms [1], [19], the entire available memory is used for reading data from disk. Even though this approach maximizes the memory utilization, it results in excessive index computation cost (index computation refers to computing the source or destination addresses of each datum during data permutation in the memory). To eliminate the index computation cost, the available memory is partitioned into two different-sized buffers (read and write buffers). Instead of performing a permutation before every write operation, only the data needed for each write operation is moved into the write buffer. This is denoted as a *collect* operation. The stride of the data access in the collect operation is constant. Thus, it can be performed using inexpensive do-loops.

If the same schedule as in the previous algorithms is used (collect operations followed by write operations), then the size of the write buffer must be $M/2$. However, in our algorithm, the utilization of the write buffer is increased using our schedule which results in a smaller write buffer. In our schedule, a write operation follows each collect operation. Since the read buffer size is less than the available memory size, the number of I/O operations is increased slightly. However, as shown in Section 6, the total execution time is reduced due to reduction in the index computation time.

5 Details of the Algorithm

Additional details of our algorithm, as well as the analysis, are presented in this section. Section 5.1 describes our technique to reduce the number of I/O operations. The overall algorithm using the read and write schedules is described first. Later, the read and write schedules are explained for the four different cases. In Section 5.2, our method to reduce the index computation time is explained.

5.1 Reducing Number of I/O Operations

We elaborate on our algorithm to reduce the number of I/O operations here (see Fig. 4). Recall that the matrix size is $N \times N$ and $N = \prod_{s=0}^{t-1} r_s$, where $r_s > 0$. Note that the order in which the product terms are computed is not important. The decomposition of N into r_s is determined by the platform architecture parameters such as the I/O cost. Choosing the values of r_s is out of the scope of this paper. Intuitively, the larger the value of r_s, the shorter the execution time is. This is because the larger value of r_s would reduce the number of stages in the algorithm. The reduction in the number of stages would reduce the memory-memory data transfer time and the index computation time. Though the number of I/O operations per stage is increased, the total number of I/O operations is reduced because of the reduced number of stages.

The algorithm consists of t stages. In each step, the data are first read into memory using read schedule $RS(s, u)$ (Line 3). The read schedule (and the write schedule later) specifies data to be read (written) in each step during a stage. The data that are read into memory are permuted based on the column number in the input matrix (Line 4).

The data is then written onto the disk using the write schedule $WS(s,u)$ (Line 5).

The RS, WS, and the permutation of data in memory are explained in detail for each of the following four cases. Case 1 and Case 2 pertain to the scenarios where as much data as the memory size can be read from the disk or written onto the disk in one I/O operation (i.e., $B \geq M$). As discussed in Case 1, our analysis shows that efficient data arrangement reduces the number of I/O operations by approximately a factor of $(r_s+1)/r_s$ compared with the previous algorithms. If $r_s \geq 8$ (Case 2), balancing the number of I/O operations reduces the total number of I/O operations by a factor of approximately $\sqrt{r_s}/2$.

If $M/r_s < B < M$ (Case 3), our algorithm provides the best performance compared with the previous algorithms with respect to the number of I/O operations. Finally, if $B \leq M/r_s$ (Case 4), the algorithm in [1] is used. The algorithm has the minimum number of I/O operations.

Note that reducing the index computation time (discussed in Section 5.2) further improves the performance in all the cases. In the following, s, $0 \leq s < t$, refers to a stage and u, $0 \leq u < N^2/M$, refers to a step.

Define R_s, $0 \leq s < t$, as follows:

$$R_s = \begin{cases} 1 & : \quad s < 0 \\ \prod_{i=0}^{s} r_i & : \quad 0 \leq s < t. \end{cases} \tag{1}$$

Also, define $r_s = 1$, if $s < 0$.

Case 1 ($B \geq M$ and $r_s < 8$): The key idea in all the cases is to arrange the data on the disk using RS and WS schedules. The $RS(s,u)$ is a set containing the row indices of the data to be read from the disk and $WS(s,u)$ is a set containing the row indices of the data to be written onto the disk during Step u in Stage s.

RS and WS for Case 1 are specified by the algorithms in Fig. 5 and Fig. 7, respectively. Examples of RS and WS are shown in Fig. 6 and Fig. 8, respectively.

If contiguous rows are read or written in a step, it can be done in one I/O operation when LM is used since, in LM, any amount of contiguous data are read or written in one I/O operation. This is true in Case 1 and Case 2. For example, in Fig. 6, in Step 0 of Stage 0, row 0 and row 1, which are contiguous, are read in one read operation. But, in Step 2 of Stage 2, row 0 and row 3 are not contiguous. Thus, they are read separately, i.e., using two read operations.

The data in the memory are permuted as in Fig. 9. $X(i)$ denotes the data at the address $i, 0 \leq i < M$, in the memory. Note that, in Fig. 9, some details are omitted for simplicity. For example, the permutations during Step 0 of Stage 1 need additional computations to identify the destination addresses. However, such additional computations are required for only a small number of permutations. Since the details are not critical for understanding the main idea of the algorithm and can be easily derived, they are omitted for simplicity.

The algorithm is illustrated in Fig. 10. The number in each small square denotes a data element. Notice that the data are in row-major order in the initial matrix in Stage 0 and in column-major order in the righthand-side matrix in Stage 2. The arrows indicate read and write operations. As discussed above, if two rows are adjacent, the data are read or written in one I/O operation.

The number of I/O operations is approximately $\frac{N^2}{M}\sum_{s=0}^{t-1} r_s$ (see Theorem 1).

A comparison of the numbers of I/O operations per step in the algorithm in [19] and our algorithm is shown in Table 1. In Case 1, Linear Model is used because M amount of data on the disk is loaded into memory in one I/O operation. Consequently, Aggarwal and Vitter's algorithm cannot be used as its basic unit of data transfer is equal to or larger than the size of memory in this particular case. However, Linear Model still works well in this case.

Several algorithms have been analyzed previously for the case where $M = 2N$. When $M = 2N$, the algorithm in [19] reduces the number of I/O operations from $2N \lg N$ in [13] to $1.5N \lg N$ by combining two read operations into one read operation. Our algorithm further reduces the number of I/O operations to approximately $N \lg N$ by combining the write operations into one write operation.

Define $f(s)$ as follows: Let $f(s)$ denote the number of additional I/O operations in a stage where $r_s = 2$ over the stage where $r_s \neq 2$. Then,

$$f(s) = \begin{cases} 1 & : \quad r_s = 2 \\ 0 & : \quad r_s \neq 2. \end{cases} \tag{2}$$

Theorem 1. *In the Linear Model, the total number of I/O operations in our algorithm for Case 1 is* $\frac{N^2}{M}\sum_{s=0}^{t-1} r_s + \alpha$, *where* $\alpha = \sum_{s=0}^{t-2} R_{s-2} f(s)$.

Proof. Consider the number of I/O operations in a step, in Stage s, $0 \leq s \leq t-1$. If $r_{s-1} \neq 2$, then one read operation and $r_s - 1$ write operations are performed. Thus, the total number of I/O operations in a step over all stages is

$$(1 + r_0 - 1) + (1 + r_1 - 1) + \ldots + (1 + r_{t-1} - 1) = \sum_{s=0}^{t-1} r_s.$$

However, if $r_{s-1} = 2$, then the ith superblock, $i = jN/R_{s-2}$ and $(j+1)N/R_{s-2} - 1, j = 0, 1, \ldots, R_{s-2} - 1$, is read in one read operation. A superblock is defined as a chunk of memory of size M/r_s in Stage s, $0 \leq s < t$. These can be read in $2R_{s-2}$ read operations. The rest of the data is read in $N^2/M - R_{s-2}$ read operations since two rows are read in one read operation. Since the number of write operations is N^2/M, the total number of I/O operations in Stage s is

$$\begin{aligned} 2R_{s-2} + N^2/M - R_{s-2} + N^2/M &= 2N^2/M + R_{s-2} \\ &= r_s N^2/M + R_{s-2}. \end{aligned}$$

This increases the number of I/O operations by R_{s-2} compared with a stage whose $r_s \neq 2$.

Let α denote the number of additional I/O operations in a stage compared with the number of operations in a stage whose r_s is not 2. Then, α is given by $2r_s N^2/M + R_{s-2} - 2r_s N^2/M = R_{s-2}$. Thus,

$$\alpha = \sum_{s=0}^{t-2} R_{s-2} f(s).$$

```
Read_schedule()
Input: N, M, t, r_0,....,r_{t-1}, R_s(0 ≤ s < t)
Output: RS(s, u)
1   for s = 0 to t − 1 step = 1
2       for u = 0 to N^2/M − 1 step = 1
3           RS(s, u) ← {};
4   for u = 0 to N^2/M − 1 step = 1
5       for j = 0 to M/N-1 step = 1
6           RS(0, u) ← RS(0, u) ∪ {(u × M/N) + j};
7   for s = 1 to t − 1 step = 1
8       u ← 0;
9       counter ← 0;
10      for i = 0 to R_{s−2}-1 step = 1
11          for j = r_{s−1} − 1 to 1 step = −1
12              for k = 0 to N/R_{s−2} − 1 step = 1
13                  if (⌊kr_{s−1}N/M⌋ − ⌊kN/M⌋) mod r_{s−1} = j
14                      Add_row(N, M, s, u, counter, i, k);
15          for k = 0 to (r_{s−1})M/N − M/(Nr_{s−1}) − 1 step = 1
16              if (⌊kr_{s−1}N/M⌋ − ⌊kN/M⌋) mod r_{s−1} = 0
17                  Add_row(N, M, s, u, counter, i, k);
18          for k = N/R_{s−2} − ⌊M/(NR_{s−1})⌋ − 1 to N/R_{s−2} − 1 step = 1
19              if (⌊kr_{s−1}N/M⌋ − ⌊kN/M⌋) mod r_{s−1} = 0
20                  Add_row(N, M, s, u, counter, i, k);
21          for k = r_{s−1}M/N − M/(Nr_{s−1}) to N/R_{s−2} − M/(NR_{s−2}) − 1 step = 1
22              if (⌊kr_{s−1}N/M⌋ − ⌊kN/M⌋) mod r_s = 0
23                  Add_row(N, M, s, u, counter, i, k);


Add_row(N, M, s, u, counter, i, k)
1   RS(s, u) ← RS(s, u) ∪ {k + Ni/R_{s−2}};
2   counter ← counter + 1;
3   if counter = M/N
4       u ← u + 1;
5       counter ← 0;
```

Fig. 5. Read schedule for Case 1.

Therefore, the total number of I/O operations in Case 1 is $\frac{N^2}{M}\sum_{s=0}^{t-1} r_s + \alpha$. □

Case 2 ($B \geq M$ and $r_s \geq 8$): In this case, the total number of I/O operations can be reduced by balancing the numbers of read and write operations. In Kaushik et al.'s algorithm, the difference between the numbers of read and write operations is large. In each step of Stage s, the number of read operations is 1 and the number of write operations is r_s. In our algorithm, we develop a technique that reduces

	s=0	s=1	s=2
$u = 0$	0,1	1,2	1,2
$u = 1$	2,3	5,6	0,3
$u = 2$	4,5	0,7	5,6
$u = 3$	6,7	3,4	4,7

Fig. 6. Read schedule for $N = 8 = \prod_{s=0}^{2} 2$, $M = 2N = 16$.

the number of write operations at the expense of an increased number of read operations. Since the absolute descriptions are similar to those in Case 1, we focus on the key ideas here. Details are given in the Appendix.

Note that a straightforward method reduces the number of write operations to $r_s - y_s, 0 \leq s \leq t-2$, where y_s is the number of the new read operations. Then, the total number of I/O operations is $(r_s - y_s) + y_s = r_s$. The total number of I/O operations is reduced by one. In our algorithm, we decrease the number of write operations to approximately r_s/z_s, where $z_s > 0$ is an integer. The optimal value of z_s is chosen later.

Define $z_{-1} = z_{t-1} = 1$. The algorithm for this case is shown in Fig. 14 (see the Appendix).

A superblock is defined as a chunk of memory of size M/r_s in stage s, $0 \leq s < t$. In the previous algorithms, each superblock is written in one disk write operation. In our algorithm, z_s superblocks are written on the disk in one write operation. The writing schedule, WS, is shown in Fig. 17 (see the Appendix). Thus, the number of write operations is reduced by a factor of z_s. Writing z_s superblocks in one I/O operation causes the data to be scattered as described below.

Note that, during the next stage (Stage $(s+1)$), r_s superblocks are read in a step. Among the r_s superblocks that are written in a step during Stage s, only one of them is read in a step during Stage $(s+1)$. Thus, r_s superblocks read in a step during Stage $(s+1)$ consist of superblocks, each of which is one of the superblocks written in different steps during Stage s.

If one superblock is written onto the disk in a write operation, as in the previous algorithms, then a superblock can be written to any location on the disk. This enables the superblocks that are read in a step during the next stage to

```
Write_schedule()
Input: N, M, t, r_0,....,r_{t-1}, R_s(0 ≤ s < t)
Output: WS(s, u)
1   for s = 0 to t − 1 step = 1
2       for u = 0 to N^2/M − 1 step = 1
3           WS(s, u) ← {};
4   for u = 0 to N^2/M − 1 step = 1
5       for j = 0 to M/N-1 step = 1
6           WS(0, u) ← WS(0, u) ∪ {(u × M/N) + j};
7   for s = 1 to t − 1 step = 1
8       u ← 0;
9       counter ← 0;
10      for i = 0 to R_{s−2}-1 step = 1
11          for j = r_{s−1} − 1 to 0 step = -1
12              for k = 0 to N/R_{s−1} − 1 step = 1;
13                  WS(s, u) ← WS(s, u) ∪ {k + N(ir_{s−1} + j)/R_{s−1}};
14                  counter ← counter + 1;
15                  if counter = M/N
16                      u ← u + 1;
17                      counter ← 0;
```

Fig. 7. Write schedule for Case 1.

	s=0	s=1	s=2
$u=0$	0,1	4,5	2,3
$u=1$	2,3	6,7	0,1
$u=2$	4,5	0,1	6,7
$u=3$	6,7	2,3	4,5

Fig. 8. Write schedule for $N = 8 = \prod_{s=0}^{2} 2$, $M = 2N = 16$.

be read in one I/O operation. However, since z_s superblocks are written in one write operation, it is impossible to write the z_s superblocks in a way to allow reading of these superblocks in one read operation during the next stage, i.e., the superblocks are "scattered."

The scattered writing causes two blocks that are read in a step during the next stage to be separated by at least $z_s - 1$ blocks. In each read operation during the next stage, in order to read the superblocks that are separated by at least $z_s - 1$ superblocks, we need to perform z_s read operations. The read schedule, RS, is shown in Fig. 15 (see the Appendix). In addition to these, if $z_s \neq 2$, one read and one write operations are performed (Line 11 and Line 15).

Permute()

Input: $N, M, s, u, r_s, X, R_s (0 \leq s < t)$

Output: X

1 **for** $j = 0$ to $r_s - 1$ $step = 1$

2 **for** $k = 0$ to $R_s - 1$ $step = 1$

3 **if** $(j,k) = (\lfloor (kr_s + j - uR_s)/R_s \rfloor, (kr_s + j - uR_s) \bmod R_s)$

4 $done(j,k) \leftarrow true$;

5 **else**

6 $done(j,k) \leftarrow false$;

7 **for** $j = 0$ to $r_s - 1$ $step = 1$

8 **for** $k = 0$ to $R_s - 1$ $step = 1$

9 **if** $done(j,k) = false$

10 Move data $X(jM/r_s + kN/R_s + lN + m)$ to $tmp(l,m)$,
$0 \leq l < M/(Nr_s), 0 \leq m < N/R_s$;

11 $(p,q) \leftarrow (j,k)$;

12 $done(j,k) \leftarrow true$;

13 $start \leftarrow (j,k)$;

14 $(j,k) \leftarrow (\lfloor (kr_s + j - uR_s)/R_s \rfloor, (kr_s + j - uR_s) \bmod R_s)$;

15 **while** $start \neq (j,k)$

16 Move data $X(jM/r_s + kN/R_s + lN + m)$ to
$X(pM/r_s + qN/R_s + lN + m)$,
$0 \leq l < M/(Nr_s), 0 \leq m < N/R_s$;

17 $done(j,k) \leftarrow true$;

18 $(p,q) \leftarrow (j,k)$;

19 $(j,k) \leftarrow (\lfloor (kr_s + j - uR_s)/R_s \rfloor, (kr_s + j - uR_s) \bmod R_s)$;

20 Move data $tmp(l,m)$ to $X(pM/r_s + qN/R_s + lN + m)$,
$0 \leq l < M/(Nr_s), 0 \leq m < N/R_s$;

Fig. 9. Permutation of data in memory for Case 1.

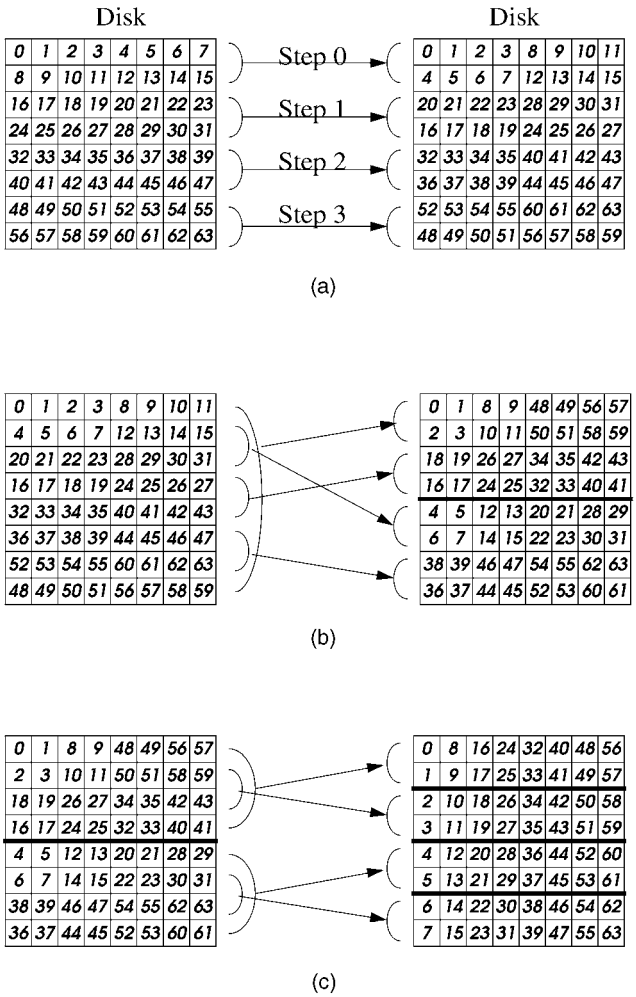


Fig. 10. An illustrative example ($N = 8 = \prod_{s=0}^{2} 2$ and $M = 16$).

Some examples of RS and WS are shown in Fig. 16 and Fig. 18, respectively (in the Appendix). The number of rows in WS is greater than M/N since there are z_s read operations in a step. Note that, in this example, the number of I/O operations is not reduced since $r_0 = r_1 = 4 < 8$. The number of I/O operations is reduced when $r_s \geq 8$, as shown in Table 2.

The optimal value of z_s is obtained as follows: Define $g(s)$ as follows:

$$g(s) = \begin{cases} 2 & : \quad z_{s-1} > 1 \\ 0 & : \quad z_{s-1} = 1. \end{cases} \tag{3}$$

The number of write operations in Stage s is $r_s/z_s + g(s)/2$ and the number of read operations to collect the data in Stage $(s+1)$ is $z_s + g(s)/2$. The total number of I/O operations is $r_s/z_s + z_s + g(s)$. The optimal value of $z_s = \sqrt{r_s}$. Then, the total number of I/O operations is $r_s/\sqrt{r_s} + \sqrt{r_s} + g(s) = 2\sqrt{r_s} + g(s)$ per step.

The total number of I/O operations performed by the algorithm in [19] and by our algorithm is compared in Table 2. The algorithm in [1] is not compared here since it is not applicable to this case.

The data in the memory are permuted as shown in Fig. 19. The total number of I/O operations for Case 2 ($M \geq B$ and $r_s \geq 8$) is given in the following theorem.

Theorem 2. *In the Linear Model, the total number of I/O operations in our algorithm is*

$$\frac{N^2}{M}\left(\sum_{s=0}^{t-2}(2\sqrt{r_s} + g(s)) + 2\right).$$

Recall that $g(s) = 2$ if $z_{s-1} \geq 2$ and $g(s) = 0$ if $z_{s-1} = 1$, where $z_s = \sqrt{r_s}$.

Proof. Consider the number of I/O operations in a step in each stage. In Stage 0, there is one read operation and $\sqrt{r_0}$ write operations. In Stage s, $1 \leq s \leq t-2$, there are $\sqrt{r_{s-1}} + g(s)/2$ read operations and $\sqrt{r_s} + g(s)/2$ write operations. In Stage $(t-1)$, there are $\sqrt{r_{t-2}} + g(t-2)/2$ read operations and one write operation. Thus, the total number of I/O operations per step over all stages is

$$\begin{aligned} &1 + \sqrt{r_0} + (\sqrt{r_0} + g(1)/2) + (\sqrt{r_1} + g(1)/2) + (\sqrt{r_1} + g(2)/2) \\ &+ \ldots + (\sqrt{r_{t-2}} + g(t-2)/2) + 1 \\ &= \sum_{s=0}^{t-2}(2\sqrt{r_s} + g(s)) + 2. \end{aligned}$$

Since the number of steps is N^2/M, the total number of I/O operations is $\frac{N^2}{M}(\sum_{s=0}^{t-2}(2\sqrt{r_s} + g(s)) + 2)$. □

TABLE 1
Number of I/O Operations in a Step

r_s	$r_s < 8$				$r_s \geq 8$		
	2	3	4	6	8	16	64
Kaushik's Algorithm [19]	3	4	5	7	9	17	65
This Paper	2	3	4	6	8	10	18
Reduction	33%	25%	20 %	17%	11%	41%	72%

TABLE 2
Number of I/O Operations in a Step

r_s	$r_s = 32$		$r_s = 128$	
	Kaushik's Algorithm	Our Algorithm	Kaushik's Algorithm	Our Algorithm
# of Read operations	1	9	1	16
# of Write operations	32	9	128	16
Total	33	18	129	32

Note that the number of I/O operations can be further reduced by using the memory more efficiently than what is shown in Theorem 2. However, the technique for more efficient memory utilization makes the algorithm description and analysis complex. Thus, only the basic idea is briefly described here. In the algorithm in [19], if z_{s-1} is not 1, the amount of data read in each read operation is $M/2$. But, the amount of data that can be read in the first read operation is M. In the next read operation, the amount of data read is $M - M/z_{s-1}$ and so on. Since the data read in one read operation is always larger than or equal to $M/2$, this method decreases the number of read operations compared with the result stated in Theorem 2.

Case 3 ($M/r_s < B < M$): This case is similar to Case 2. The only difference is in the size of the superblock. Our algorithm relaxes the restriction ($r_s \leq M/B$) that was imposed in [1]. If the restriction does not hold, i.e., $r_s > M/B$, then the number of I/O operations increases by a factor of $\lceil Br_s/M \rceil$. In our algorithm, we can increase the value of r_s to be larger than M/B without increasing the number of I/O operations so that the number of stages is decreased.

The algorithm is the same as in Case 2 (Fig. 14). Algorithms for computing RS and WS are shown in Fig. 20 and Fig. 21, respectively. The entries in RS and WS are superblock indices (In Case 2, the entries in RS and WS contain row indices). A superblock is defined as a chunk of memory of size M/r_s in Stage s, $0 \leq s < t$.

The total number of I/O operations in each step over all stages is

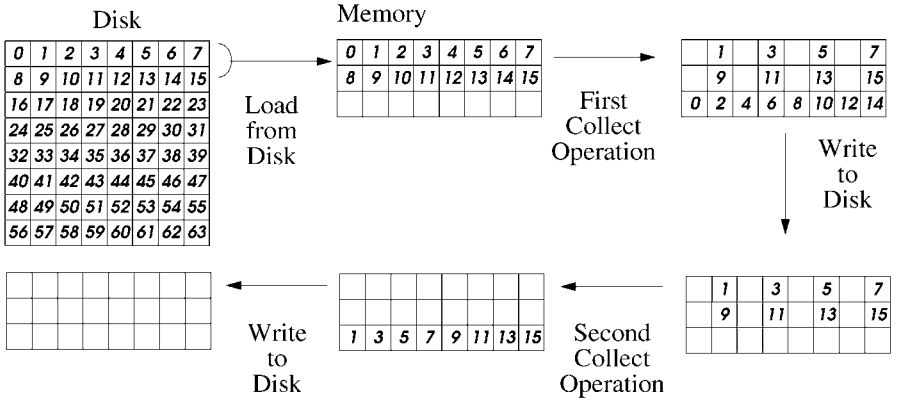
TABLE 3
Comparison of the Number of I/O Operations for $D = 1$

	Linear Model (LM)		Parallel Disk Model (PDM)	
Algorithm	$r_s < 8$	$r_s \geq 8$	$B \leq \frac{M}{r_s}$	$B > \frac{M}{r_s}$
Aggarwal[1]	-	-	$\frac{2N^2}{B} \lg_{\frac{M}{B}} \min(\frac{N^2}{B}, B)$	-
Kaushik[19]	$\frac{N^2}{M} \sum_{s=0}^{t-1} (1 + r_s)$	$\frac{N^2}{M} \sum_{s=0}^{t-1} (1 + r_s)$	$\frac{2N^2}{B} \lg_{\frac{M}{B}} \min(\frac{N^2}{B}, B)$	$\frac{N^2}{B} \sum_{s=0}^{t-1} (\frac{Br_s}{M} + 1)$
This Paper	$\frac{N^2}{M} \sum_{s=0}^{t-1} r_s$	$\frac{N^2}{M} \sum_{s=0}^{t-2} 2\sqrt{r_s}$	$\frac{2N^2}{B} \lg_{\frac{M}{B}} \min(\frac{N^2}{B}, B)$	$\frac{2N^2 t}{B} + \frac{N^2}{M} \sum_{s=0}^{t-2} (\min(\frac{M}{B}, \sqrt{r_s}) + \sqrt{r_s})$

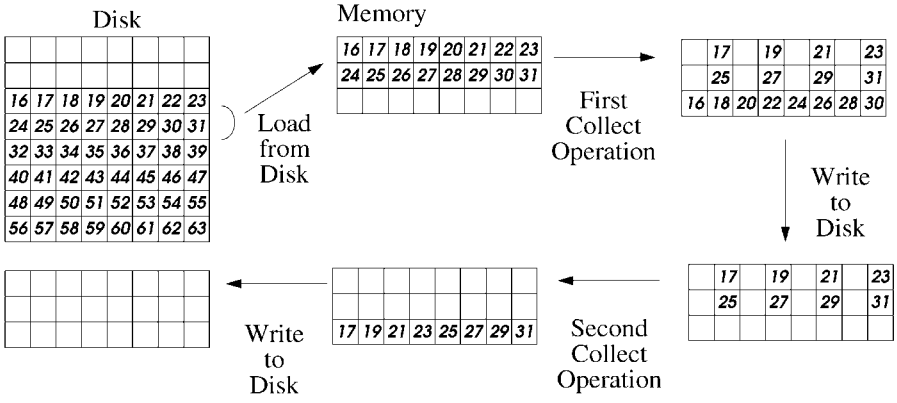
s, $0 \leq s < t$, refers to the stage. For clarity, smaller terms are not included in the expression. See Section 5.1 for details.

```
1  for s = 0 to t - 1
2      for step = 0 to N^2/M_r - 1
3          Read data from disk;
4          for i = 0 to r_s/z_s - 1
5              for j = 0 to z_s - 1
6                  Move (iz_s + j)^th superblock to write buffer;
7              Write data in write buffer to disk;
```

Fig. 11. Pseudocode for our algorithm to reduce index computation time.



(a)



(b)

Fig. 12. An illustration of the collect operation. (a) Step 0. (b) Step 1.

$$
\begin{aligned}
&(M/B + \max(r_0/z_0, M/B)) + \max(M/B, \min(\sqrt{r_0}M/B, z_0)) \\
&+ g(1) + \max(r_1/z_1, M/B) + \max(M/B, \min(\sqrt{r_1}M/B, z_1)) \\
&+ g(2) + \max(r_2/z_2, M/B) + \ldots \\
&+ \max(M/B, \min(\sqrt{r_{t-3}}M/B, z_{t-3})) + g(t-1) \\
&+ \max(r_{t-2}/z_{t-2}, M/B) + \max(M/B, \min(\sqrt{r_{t-2}}M/B, z_{t-2})) \\
&+ g(t) + M/B \\
&= 2M/B + \sum_{s=0}^{t-2} (\max(M/B, \min(\sqrt{r_s}M/B, z_s)) \\
&+ g(s+1) + \max(r_s/z_s, M/B)).
\end{aligned}
$$

The optimal value of the z_s can be calculated as in Case 2 ($\frac{M}{r_s} < B < M$, $0 \le s \le t-1$). The optimal value of the z_s is $\min(\sqrt{r_s}, Br_s/M)$.

Theorem 3. *In the Parallel Disk Model, the total number of I/O operations in our algorithm is*

$$\frac{2N^2t}{B} + \frac{N^2}{M}\sum_{s=0}^{t-2}\left(\min\left(\frac{M}{B}, \sqrt{r_s}\right) + \sqrt{r_s} + g(s+1)\right).$$

Proof. The total number of I/O operations in each step over all stages is given by:

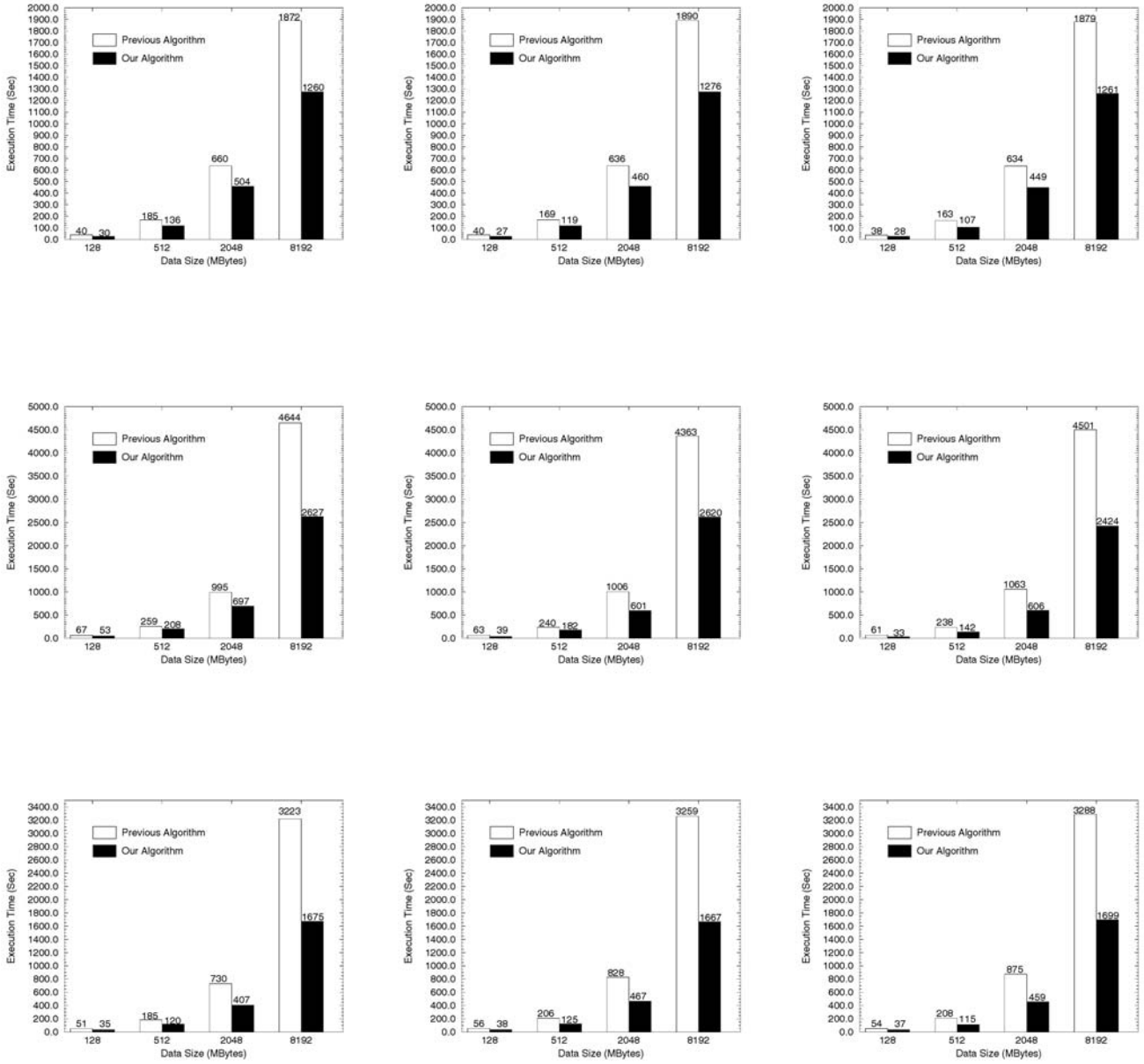


Fig. 13. Experimental results (average execution time; top row: PC Pentium, center row: Sun Enterprise, bottom row: SGI R12000; left column: $M = 16$ MBytes, right column: $M = 64$ MBytes.

$$2M/B + \sum_{s=0}^{t-2}(\max(M/B, \min(\sqrt{r_s}M/B, z_s)) + g(s+1) + \max(r_s/z_s, M/B)).$$

The optimal value of z_s is $\min(\sqrt{r_s}, B/M)$. Thus, if $\sqrt{r_s} > B/M$, then the total number of the I/O operations is $\frac{2N^2t}{B} + \frac{N^2}{M}\sum_{s=0}^{t-2}(2\sqrt{r_s} + g(s+1))$.

If $\sqrt{r_s} \leq B/M$, then the total number of the I/O operations is $\frac{2N^2t}{B} + \frac{N^2}{M}\sum_{s=0}^{t-2}(\frac{M}{B} + \sqrt{r_s} + g(s+1))$.

Thus, the total number of I/O operations is $\frac{2N^2t}{B} + \frac{N^2}{M}\sum_{s=0}^{t-2}(\min(\frac{M}{B}, \sqrt{r_s}) + \sqrt{r_s} + g(s+1))$. □

Case 4 ($B \leq M/r_s$): In this case, the algorithm in [1] is used. The algorithm has the minimum number of I/O operations.

A comparison of the algorithms with respect to the number of I/O operations is shown in Table 3.

5.2 Reducing Index Computation Time

In the previous algorithms, the available memory is fully utilized to reduce the number of I/O operations. In other words, in a read operation, as much data as the size of the available memory are read from the disk. After reading the data, permuting the data within the memory requires the destination location of each data element to be computed. This results in a large index computation time.

To reduce the total execution time, we eliminate the expensive index computation time by using the algorithm shown in Fig. 11. In our algorithm, we partition the memory into two different-sized buffers: one of size M_r, which is used as a read buffer, and the other of size M_w, which is used as a write buffer, such that $M = M_r + M_w$. The read

TABLE 5
Experimental Results on Sun UltraSPARC-III

Data Size (MBytes)		Memory Size = 16 MBytes			Memory Size = 32 MBytes			Memory Size = 64 MBytes		
		Previous	Our	Speedup	Previous	Our	Speedup	Previous	Our	Speedup
128	Min	64	51	1.25	62	38	1.63	56	31	1.81
	Avg	67	53	1.26	63	39	1.62	61	33	1.85
	Max	68	54	1.26	64	40	1.60	63	35	1.80
512	Min	257	202	1.27	228	174	1.31	235	136	1.73
	Avg	259	208	1.25	240	182	1.32	238	142	1.68
	Max	260	213	1.22	246	184	1.34	244	150	1.63
2048	Min	991	684	1.45	998	597	1.67	1050	600	1.75
	Avg	995	697	1.43	1006	601	1.67	1063	606	1.75
	Max	999	708	1.41	1015	606	1.67	1067	608	1.75
8192	Min	4639	2581	1.80	4285	2613	1.64	4394	2417	1.82
	Avg	4644	2627	1.77	4363	2620	1.67	4501	2424	1.86
	Max	4653	2654	1.75	4432	2629	1.69	4591	2433	1.89

buffer is partitioned into r_s superblocks. The read buffer is used for reading data from disk. After reading the data, there are r_s/z_s sets of collect and write operations (see Section 4.2.2). In each collect operation, data in z_s superblocks are collected into the write buffer and this operation is repeated r_s/z_s times (Line 5). The sizes of the write and read buffers are determined to be $Mz_s/(r_s + z_s)$ and $Mr_s/(r_s + z_s)$, respectively.

Fig. 12 shows an example of the collect operation. In this example, $N = 8$, $M = 24$, $r_s = 2$, $z_s = 1$. Thus, $M_r = 24 * 2/3 = 16$ and $M_w = 24/3 = 8$. In the figure, the first two steps during a stage are shown to illustrate the collect operation. Initially, two rows of data are loaded into the memory from the disk. Note that, even though the memory can hold three rows, only two rows are loaded since $M_r = 16$. The remaining space ($M_w = 8$) is utilized by the write buffer. After loading data into the memory, the data for the first write operation (0, 2, 4, ..., 14) are collected in the write buffer. Then, the data in the write buffer is written onto the disk. Since the write buffer is empty now, the data for the second write operation (1, 3, 5, ..., 15) are collected into the write buffer, which is written onto the disk after collection. This is repeated until all the data in the memory are written onto the disk. In

TABLE 6
Experimental Results on PC (Pentium III, 733MHz)

Data Size (MBytes)		Memory Size = 16 MBytes			Memory Size = 32 MBytes			Memory Size = 64 MBytes		
		Previous	Our	Speedup	Previous	Our	Speedup	Previous	Our	Speedup
128	Min	40	30	1.33	39	27	1.44	38	27	1.41
	Avg	40	30	1.33	40	27	1.48	38	28	1.36
	Max	41	31	1.32	40	28	1.43	40	29	1.38
512	Min	183	134	1.37	168	116	1.45	162	106	1.53
	Avg	185	136	1.36	169	119	1.42	163	107	1.52
	Max	186	141	1.32	169	121	1.40	164	109	1.50
2048	Min	651	502	1.30	632	453	1.40	631	446	1.41
	Avg	660	504	1.31	636	460	1.38	634	449	1.41
	Max	667	507	1.32	647	476	1.36	640	458	1.40
8192	Min	1868	1259	1.48	1890	1274	1.48	1877	1259	1.49
	Avg	1872	1260	1.49	1890	1276	1.48	1879	1261	1.49
	Max	1882	1261	1.49	1891	1279	1.48	1880	1264	1.49

TABLE 7
Experimental Results on SGI R12000

Data Size (MBytes)		Memory Size = 16 MBytes			Memory Size = 32 MBytes			Memory Size = 64 MBytes		
		Previous	Our	Speedup	Previous	Our	Speedup	Previous	Our	Speedup
128	Min	50	31	1.61	54	35	1.54	52	34	1.53
	Avg	51	35	1.46	56	38	1.47	54	37	1.45
	Max	53	37	1.43	60	40	1.50	55	40	1.38
512	Min	183	118	1.55	204	124	1.65	208	115	1.81
	Avg	185	120	1.54	206	125	1.65	208	115	1.81
	Max	185	121	1.53	206	126	1.63	209	115	1.82
2048	Min	729	405	1.80	824	464	1.78	867	455	1.91
	Avg	730	407	1.79	828	467	1.77	875	459	1.91
	Max	730	409	1.78	834	469	1.78	889	461	1.93
8192	Min	3228	1655	1.95	3257	1659	1.96	3286	1697	1.94
	Avg	3223	1675	1.92	3259	1667	1.96	3288	1699	1.94
	Max	3265	1697	1.92	3260	1684	1.94	3293	1713	1.92

```
1   for s = 0 to t-1 step = 1
2       for u = 0 to N^2/M-1 step = 1
3           if z_{s-1} = 1
4               Read M units of data from disk using read schedule RS(s, u);
5               Permute the data in memory;
6               Write M units of data onto disk using write schedule WS(s, u);
7           else
8               for i = 0 to z_{s-1} - 1 step = 1
9                   Read M/2 units of data from disk using read schedule RS(s, u);
10                  Permute the data in memory;
11              Write M/2 units of data to a buffer on disk;
12              for i = 0 to z_{s-1} - 1 step = 1
13                  Read M/2 units of data from disk using read schedule RS(s, u);
14                  Permute the data in memory;
15              Read M/2 units of data from disk in the buffer on disk;
16              Permute data in memory;
17              Write M units of data to disk using write schedule WS(s, u);
```

Fig. 14. Matrix transposition for Case 2 and Case 3.

Step 1, the data (16, 17, 18, ..., 31) are loaded into the memory and the above operations are repeated.

In a collect operation, the data in z_s superblocks are collected using do-loops since the data access stride is constant. In each do-loop, the required computations are simple additions. Note that, in the previous algorithms [1], [19], the computations to permute the data consist of both index computation and addition. In our algorithm, since only addition operations are used to collect data to the write buffer, the index computation is eliminated.

The collected data in the write buffer is written onto the disk in a write operation (Line 6). Even though the number of I/O operations increases by a factor of M/M_r, the total

```
Read_schedule()
Input: N, M, t, r_0,...,r_{t-1}, z_s, R_s(0 ≤ s < t)
Output: RS(s, u)
1   for s = 0 to t-1 step = 1
2       for u = 0 to N^2/M-1 step = 1
3           RS(s, u) ← {};
4   for s = 0 to t − 1 step = 1
5       u ← 0;
6       counter ← 0;
7       for i = 0 to R_{s−1}/z_{s−1}-1 step = 1
8           for j = 0 to z_{s−1} − 1 step = 1
9               for k=0 to N z_{s−1}/R_{s−1} − 1 step = 1
10                  RS(s, u) ← RS(s, u) ∪ {i N z_{s−1}/R_{s−1} + k};
11                  counter ← counter + 1;
12                  if counter = M z_{s−1}/N
13                      counter ← 0;
14                      u ← u + 1;
```

Fig. 15. Read schedule for Case 2.

execution time reduces significantly, as shown by our experiments.

6 Experimental Results

We implemented the algorithms on an SGI (R12000, 300 MHz), a Sun Enterprise system (UltraSPARC-III, 750 MHz), and a PC platform (Pentium III, 733 MHz) at the University of Southern California. For the sake of comparison, Kaushik et al.'s algorithm described in Section 3.3 was also implemented. The results are shown in Fig. 13 and Table 5, Table 6, and Table 7.

In our experiments, we observed that Aggarwal and Vitter's algorithm, described in Section 3.2, has the same total execution time as Kaushik et al.'s algorithm. Even though the two algorithms perform the needed permutation using different methods and the permuted data are different, the permutation times are the same. If, in Stage s, $0 \leq s < t$, the block size is smaller than M/r_s, then both algorithms require the same total execution time, where t is the number of stages. The total execution time is different for the two algorithms when the amount of data transferred in one I/O operation is smaller than B. The amount of the data transferred in one I/O operation in our experiments ranges from 128 KBytes to 8 MBytes and the typical size of B in state-of-the-art platforms is 4 KBytes. Thus, the performance of the two algorithms is the same in our experiments. Therefore, in Fig. 13 and Table 5, Table 6, and Table 7, the execution time reported under the heading "Previous" refers to both these algorithms.

	s=0	s=1
$u = 0$	0,1,2,3	0,1,2,3,4,5,6,7
$u = 1$	4,5,6,7	0,1,2,3,4,5,6,7
$u = 2$	8,9,10,11	8,9,10,11,12,13,14,15
$u = 3$	12,13,14,15	8,9,10,11,12,13,14,15

Fig. 16. Read schedule for $N = 16 = \prod_{s=0}^{1} 4$, $M = 32$, $z_0 = 2$.

The amount of main memory allocated to the data was varied from 16 MBytes to 64 MBytes and the data size was varied from 128 MBytes to 8 GBytes. For each parameter (memory and data size) setting, the algorithms were executed five times and the maximum, average, and minimum values were calculated. The reported times are wall-clock times and the unit is second. The speedup of our algorithm over the previous algorithms was calculated for each parameter setting. The results of our experiments are shown in Fig. 13. Additional results are shown in Table 5, Table 6, and Table 7. The results show that our algorithm reduces the execution time by up to 50 percent.

To understand the effect of the system parameters on the speedup, we define the following terms: Let T_d denote disk I/O time , T_m denote the memory-memory data transfer time, T_c denote the index computation time, and T_a denote the additional disk I/O time required by our algorithm compared with the previous algorithms. Also, we define R_c and R_a ratios as $R_c = T_c/(T_d + T_m)$ and $R_a = T_a/(T_d + T_m)$.

```
Write_schedule()
Input: N, M, t, r_0,...,r_{t-1}, z_s, R_s(0 ≤ s < t)
Output: RS(s, u)
1   for s = 0 to t-1 step = 1
2       for u = 0 to N^2/M-1 step = 1
3           WS(s, u) ← {};
4   for s = 0 to t − 1 step = 1
5       for i = 0 to R_{s−1}-1 step = 1
6           for j = 0 to N^2/(R_{s−1}M) − 1 step = 1
7               for k = 0 to r_s/z_s − 1 step = 1
8                   for l = 0 to Mz_s/(Nr_s) − 1 step = 1
9                       WS(s, iN^2/(R_{s−1}M) + j)
                            ← WS(s, iN^2/(R_{s−1}M) + j) ∪ {iN/R_{s−1} + jMz_s/(Nr_s) + kNz_s/R_s + l};
```

Fig 17. Write schedule for Case 2.

Note that R_a is usually much smaller than one as the sum of the disk I/O and memory-memory data transfer times is larger than the additional disk I/O time incurred due to the partitioning of memory into two buffers. Then, the speedup of the new algorithm compared with the previous algorithm is

$$\text{Speedup} = \frac{T_d + T_m + T_c}{T_d + T_m + T_a} = \frac{1 + R_c}{1 + R_a} \approx 1 + R_c.$$

The equation shows that the speedup is a function of the two ratios, R_a and R_c. Note that the parameters of the disk system, memory, and the CPU are all accomodated in the two simple ratios. And, also, accurate measurement of these parameters is difficult to perform. For example, if the disk access time reduces, then R_c increases and, if the processor speed increases, then R_c decreases. The values of the R_c for the systems on which experiments were performed have been measured to be 1.64 (SGI), 1.29 (Sun), and 0.66 (Pentium).

The speedup on the SGI is the highest and that on the Pentium is the lowest among the three machines as the R_c of the SGI is the largest and that of the Pentium is the smallest.

Current operating systems do not support file sizes larger than 2 GB. To perform experiments for the file sizes as large as 8 GB, we partitioned the file into four 2 GB files. Then, based on the address, appropriate files are accessed during the execution of the algorithm. By using this technique, we were able to perform experiments of data sizes larger than 2 GB for the machines considered in this paper. However, there exist slight variations in the obtained results. For example, the execution time obtained for 8 GB file sizes is much smaller than four times the execution time obtained for 2 GB file sizes for some operating systems. We believe the difference is due to the optimization of the disk system by the operating system, for multiple file accesses.

	s=0	s=1
$u = 0$	0,1,8,9	0,1,2,3
$u = 1$	2,3,10,11	4,5,6,7
$u = 2$	4,5,12,13	8,9,10,11
$u = 3$	6,7,14,15	12,13,14,15

Fig. 18. Write schedule for $N = 16 = \prod_{s=0}^{1} 4$, $M = 32$, $z_0 = 2$.

7 Further Extensions

Our matrix transpose algorithm can be easily extended to a parallel disk version by using the same method as in [1]. The algorithm in [1] uses a RAID-type disk system that has D disks. The disks provide D parallel accesses. By allocating N/D columns to each disk and accessing D disks simultaneously, the disk data transfer time is reduced by a factor of D.

To reduce the number of I/O operations, the algorithm in Fig. 4 can be used to employ the two techniques proposed in this paper: efficient read and write schedules and balancing the number of I/O operations.

To reduce the index computation time, the algorithm in Fig. 11 is used: The available memory is partitioned into two buffers, the permutation is replaced by collect operations, and the collect operations and write operations are scheduled to maximize the utilization of the available memory.

8 Conclusion

In this paper, we presented an efficient algorithm for large-scale matrix transposition. Contrary to previous works that have focused only on reducing the number of I/O operations, we identified the major costs in the state-

```
Permute()
Input: N, M, s, r_s, X, R_s(0 ≤ s < t)
Output: X
1   for j = 0 to r_s − 1 step = 1
2       for k = 0 to R_s − 1 step = 1
3           if (j,k) = (⌊(kr_s + j)/R_s⌋, (kr_s + j) mod R_s))
4               done(j,k) ← true;
5           else
6               done(j,k) ← false;
7   for j = 0 to r_s − 1 step = 1
8       for k = 0 to R_s − 1 step = 1
9           if done(j,k) = false
10              Move data X(jM/r_s + kN/R_s + lN + m) to tmp(l,m),
                 where 0 ≤ l < M/(Nr_s), 0 ≤ m < N/R_s;
11              (p,q) ← (j,k);
12              done(j,k) ← done;
13              start ← (j,k);
14              (j,k) ← ⌊((kr_s + j)/R_s⌋, k ← (kr_s + j) mod R_s);
15              while start ≠ (j,k)
16              Move data X(jM/r_s + kN/R_s + lN + m) to
                X(pM/r_s + qN/R_s + lN + m),
                where 0 ≤ l < M/(Nr_s), 0 ≤ m < N/R_s;
17                  done(j,k) ← true;
18                  (p,q) ← (j,k);
19                  (j,k) ← (⌊(kr_s + j)/R_s⌋, k ← (kr_s + j) mod R_s);
20              Move data tmp(l,m) to X(pM/r_s + qN/R_s + lN + m),
                where 0 ≤ l < M/(Nr_s), 0 ≤ m < N/R_s;
```

Fig. 19. Permutation of data in memory for Case 2.

of-the-art computing platforms in performing transpose and the overall cost was reduced.

The generality of the main ideas lends itself to applicability to other algorithms having recursive structures that operate on large data sets.

APPENDIX

This section contains pseudocode for the matrix transposition and read and write schedules for Cases 2 and 3 of our algorithm, shown in Figs. 14, 15, 16, 17, 18, 19, 20, and 21.

ACKNOWLEDGMENTS

The authors would like to thank the US Army Engineering Research and Development Center and Major Shared Resource Center staff for their support in conducting the experiments. They would also like to thank Bharani Thiruvengadam for his assistance in preparing this manuscript. This work was funded by the US Department of Defense (DoD) High Performance Modernization Program ERDC Major Shared Resource Center through Programming Environment and Training (PET) supported by Contract Number DAHC 94-96-C0002 and Subcontract

```
Read_schedule()
Input: N, M, t, r_0,...,r_{t-1}, z_s, R_s(0 ≤ s < t)
Output: RS(s, u)
1   for s = 0 to t-1 step = 1
2       for u = 0 to N^2/M-1 step = 1
3           RS(s, u) ← {};
4   for s = 0 to t − 1 step = 1
5       u ← 0;
6       counter ← 0;
7       for i = 0 to R_{s−1}/z_{s−1}-1 step = 1
8           for j = 0 to z_{s−1} − 1 step = 1
9                   for k=0 to N^2 z_{s−1}/(BR_{s−1} − 1) step = 19
10                  counter ← counter + 1;
11                  if counter = M z_{s−1}/B
12                      counter ← 0;
13                      u ← u + 1;
```

Fig. 20. Read schedule for Case 3.

```
Write_schedule()
Input: N, M, t, r_0,...,r_{t-1}, z_s, R_s(0 ≤ s < t)
Output: RS(s, u)
1   for s = 0 to t-1 step = 1
2       for u = 0 to N^2/M-1 step = 1
3           WS(s, u) ← {};
4   for s = 0 to t − 1 step = 1
5       for i = 0 to R_{s−1}-1 step = 1
6           for j = 0 to N^2/(R_{s−1}M) − 1 step = 1
7               for k = 0 to r_s/z_s − 1 step = 1
8                   for l = 0 to M z_s/(B r_s) − 1 step = 1
9                       WS(s, iN/(R_{s−1}M) + j)
                            ← WS(s, iN/(R_{s−1}M) ∪ {iN^2/(BR_{s−1}) + kM z_s/(BR_s) + l};
```

Fig. 21. Write schedule for Case 3.

Number NRC-CR-98-0002. Most of this work was performed while Jinwoo Suh was a doctoral student at the University of Southern California. The current work of the Dr. Suh is funded by the US Defense Advanced Research Projects Agency (DARPA) through the Air Force Research Laboratory, USAF, under agreement number F30602-99-1-0521. Views, opinions, and/or findings contained in this report are those of the authors and should not be construed as an official DoD, DARPA, or Air Force Research Laboratory position, policy, or decision unless so designated by other official documentation.

REFERENCES

[1] A. Aggarwal and J.S. Vitter, "The Input/Output Complexity of Sorting and Related Problems," *Comm. ACM,* vol. 31, no. 9, pp. 1116-1127, 1988.

[2] M.B. Ari, "On Transposing Large $2^n \times 2^n$ Matrices," *IEEE Trans. Computers,* vol. 28, no. 1, pp. 72-75, Jan. 1979.

[3] R. Bernecky, "Sonar Beamforming Challenge Problems," presented at DARPA/ITO Embeddable Systems PI Meeting, June 1996.

[4] L. Carter, J. Ferrante, and S.F. Hummel, "Hierarchical Tiling for Improved Superscalar Performance," *Proc. Int'l Parallel Processing Symp. (IPPS '95),* 1995.

[5] P.M. Chen, E.K. Lee, G.A. Gibson, R.H. Katz, and D.A. Patterson, "RAID: High Performance, Reliable Secondary Storage," *ACM Computing Surveys,* vol. 26, no. 2, pp. 145-185, June 1994.

[6] T.H. Cormen, "Virtual Memory for Data-Parallel Computing," PhD Thesis, Massachussetts Inst. of Technology, MIT/LCS/TR-559, 1992.

[7] T.H. Cormen and M. Hirschl, "Early Experiences in Evaluating the Parallel Disk Model with the ViC* Implementation," *Parallel Computing,* vol. 23, nos. 4-5, pp. 571-600, June 1997.

[8] T.H. Cormen, T. Sundquist, and L.F. Wisniewski, "Asymptotically Tight Bounds for Performing BMMC Permutations on Parallel Disk Systems," *SIAM J. Computing,* vol. 28, no. 1, pp. 105-136, 1994.

[9] DARPA, "Embeddable Systems Home page,"http://www.darpa.mil/ito/ResearchAreas.html, 2000.

[10] DARPA, http://www.darpa.mil/ito/research/dis/index.html, 2000.

[11] L.G. Delcaro and G.L. Sicuranza, "A Method on Transposing Externally Stored Matrices," *IEEE Trans. Computers,* vol. 23, no. 9, pp. 967-970, 1974.

[12] D.E. Dudgen and R.M. Mersereau, *Multidimensional Signal Processing.* Prentice Hall, 1984.

[13] J.O. Eklundh, "A Fast Computer Method for Matrix Transposing," *IEEE Trans. Computers,* vol. 20, no. 7, pp. 801-803, 1972.

[14] R.W. Floyd, "Permuting Information in Idealized Two-Level Storage," *Complexity of Computer Computations,* pp. 105-109, Plenum, 1972.

[15] R.A. Games, "Benchmarking Methodology for Real-Time Embedded Scalable High Performance Computing," MITRE Technical Report MTR 96B0000010, Mar. 1996.

[16] K. Hwang and Z. Xu, "Scalable Parallel Computers for Real-Time Signal Processing," *IEEE Signal Processing Magazine,* vol. 13, no. 4, pp. 50-66, July 1979.

[17] M. Kallahalla and P. Varman, "Optimal Read-Once Parallel Disk Scheduling," *Proc. ACM Workshop I/O in Parallel and Distributed Systems,* Apr. 1999.

[18] M. Kallahalla and P. Varman, "An Improved Parallel Prefetching Algorithm," *Proc. Int'l Conf. High Performance Computing,* Dec. 1998.

[19] S.D. Kaushik, C.-H. Huang, J.R. Johnson, R.W. Johnson, and P. Sadayappan, "Efficient Transposition Algorithms for Large Matrices," *Proc. Supercomputing,* 1993.

[20] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing.* Benjamin/Cummings, 1994.

[21] M. Lee, W. Liu, and V.K. Prasanna, "A Mapping Methodology for Designing Software Task Pipelines for Embedded Signal Processing," *Proc. Third Int'l Workshop Embedded HPC Systems and Applications (EHPC '98), at the 12th Int'l Parallel Processing Symp. (IPPS '98), and the Ninth Symp. Parallel and Distributed Processing (SPDP '98),* Apr. 1979.

[22] Y.W. Lim and V.K. Prasanna, "Scalable Portable Implementations of Space-Time Adaptive Processing," *Proc. 10th Int'l Conf. High Performance Computers,* June 1996.

[23] Y.W. Lim, P.B. Bhat, and V.K. Prasanna, "Efficient Algorithms for Block-Cyclic Redistribution of Arrays," *Algorithmica,* vol. 24, pp. 298-330, 1999.

[24] H. Park, J. Suh, V.K. Prasanna, and M. Ung, "Parallel Implementation of 2D FFT on High Performance Computing Platforms," *Proc. DoD HPC User's Conf. '98,* June 1998.

[25] H.K. Ramapriyan, "A Generalization of Eklundh's Algorithm for Transposing Large Matrices," *IEEE Trans. Computers,* vol. 24, no. 12, pp. 1221-1226, Dec. 1975.

[26] J.C. Sheperdson and H.E. Sturgis, "Computability of Recursive Functions," *J. ACM,* vol. 10, pp. 217-255, 1963.

[27] J. Suh and V.K. Prasanna, "Portable Implementation of Real Time Signal Processing Benchmarks on HPC Platforms," *Proc. Int'l Workshop Applied Parallel Computing in Large Scale Scientific and Industrial Problems '98,* June 1998.

[28] J.S. Vitter and E.A.M. Shriver, "Algorithms for Parallel Memory I: Two-Level Memories," *Algorithmica,* vol. 12, nos. 2-3, pp. 110-147, 1994.

Jinwoo Suh received the PhD degree in electrical engineering from the University of Southern California in 1999, the MS degree in electrical engineering from the Korea Advanced Institute of Science and Technology (KAIST), Seoul, Korea, in 1990, and the BS degree in electronic engineering from Dongguk University, Seoul, Korea, in 1988. He is a computer scientist in the Dynamic Systems Division at the University of Southern California/Information Sciences Institute (USC/ISI) in Arlington, Virginia. While he was an engineer at the research center of the DAEWOO Electronics Co., Ltd, Seoul, Korea, from 1990 to 1994, he developed several research prototypes for High Definition TV (HDTV) and other commercial products. His research interests include parallel processing, algorithm design, real-time processing, and computer architecture. He is a member of the IEEE and IEEE Computer Society.

Viktor K. Prasanna received the BS degree in electronics engineering from the Bangalore University, the MS degree from the School of Automation, Indian Institute of Science, and the PhD degree in computer science from Pennsylvania State University. He is a professor of electrical engineering and computer science at the University of Southern California (USC). His research interests include parallel computation, computer architecture, VLSI computations, high performance computing for signal processing, image processing, and vision. He is also a member of the US National Science Foundation (NSF)-supported Integrated Media Systems Center (IMSC) and an associate member of the Center for Applied Mathematical Sciences (CAMS) at USC. He has published extensively and consulted for industries in the above areas. He is the Steering Committee cochair of the International Parallel & Distributed Processing Symposium (IPDPS) (merged IEEE International Parallel Processing Symposium (IPPS) and Symposium on Parallel and Distributed Processing (SPDP)). He is the Steering Committee chair of the International Conference on High Performance Computing (HiPC). He serves on the editorial boards of the *Journal of Parallel and Distributed Computing*, *IEEE Transactions on VLSI Systems*, and *IEEE Transactions on Computers*. He was the founding chair of the IEEE Computer Society's Technical Committee on Parallel Processing. He is a fellow of the IEEE.

A Performance Analysis of PIM, Stream Processing, and Tiled Processing on Memory-Intensive Signal Processing Kernels

Jinwoo Suh, Eun-Gyu Kim, Stephen P. Crago, Lakshmi Srinivasan, and Matthew C. French
University of Southern California/Information Sciences Institute
3811 N. Fairfax Drive, Suite 200, Arlington, VA 22203
{jsuh, eungyu, crago, lakshmi, mfrench}@isi.edu

Abstract

Trends in microprocessors of increasing die size and clock speed and decreasing feature sizes have fueled rapidly increasing performance. However, the limited improvements in DRAM latency and bandwidth and diminishing returns of increasing superscalar ILP and cache sizes have led to the proposal of new microprocessor architectures that implement processor-in-memory, stream processing, and tiled processing. Each architecture is typically evaluated separately and compared to a baseline architecture. In this paper, we evaluate the performance of processors that implement these architectures on a common set of signal processing kernels.

The implementation results are compared with the measured performance of a conventional system based on the PowerPC with Altivec. The results show that these new processors show significant improvements over conventional systems and that each architecture has its own strengths and weaknesses.

1. Introduction

Microprocessor performance has been doubling every 18-24 months for many years [7]. This increase has been possible because die size has increased and feature size has decreased. However, the increasing die size combined with fast clock speeds have made the maximum distance as measured in clock cycles between two points on a processor longer.

To solve this problem, pipelining has been used widely. However, increasing pipeline depth increases various latencies, including cache access and branch prediction penalties, and increases the complexity of processor design. Techniques for exploiting ILP without exposing parallelism to the instruction set have also reached a point of diminishing returns.

Another problem in the recent processors is the growing gap between the processor speed and memory speed. The performance improvement of microprocessors has not been matched by DRAM (main memory) latencies, which have only improved by 7% per year [7], or pin bandwidths. These growing gaps have created a problem for data-intensive applications.

To bridge these growing gaps, many methods have been proposed such as caching, prefetching, and multithreading. However, these methods provide limited performance improvement and can even hinder performance for data-intensive applications. Caching has been the most popular memory latency tolerating technique [10][12]. Caching increases performance by utilizing temporal and spatial locality, but it is not useful for many data-intensive applications since many of them do not show such locality [11].

Recently, several architectural approaches have been explored that promise to hide memory latency for applications that include data-intensive applications while improving scalability. This study is an attempt to demonstrate and compare some of the advantages and disadvantages of processor-in-memory (PIM), streaming, and tiled architectures approaches by implementing a common set of memory-intensive signal processing kernels.

We implemented the corner turn, beam steering, and coherent side-lobe canceller (CSLC) kernels and measured the performance using cycle accurate simulators developed by each architecture group.

The rest of the paper is organized as follows. In Chapter 2, a PIM, a stream processor, and a tile-based processor are briefly described. Chapter 3 describes the three kernels we implemented: the corner turn, coherent side-lobe canceller, and beam steering. Also, the techniques that we used to exploit each platform are described. In Chapter 4, the implementation results and analysis are shown. Chapter 5 concludes the paper.

2. VIRAM, IMAGINE, and RAW

In this section, the VIRAM, Imagine, and Raw chips are briefly described. We also describe the performance models that will be used to understand performance of the application kernels.

2.1 VIRAM

In conventional systems, the CPU and memory are implemented on different chips. Thus, the bandwidth between CPU and memory is limited since the data must be transferred through chip I/O pins and copper wires on a PCB. Furthermore, much of the internal structure of DRAM, which could be exploited if exposed, is hidden because of the bandwidth limitation imposed by the pins.

Processor-In-Memory (PIM) technology is a method for closing the gap between memory speed and processor speed for data intensive applications. PIM technology integrates a processor and DRAM on the same chip. The integration of memory and processor on the same chip has the potential to decrease memory latency and increase the bandwidth between the processor and memory. PIM technology also has the potential to decrease other important system parameters such as power consumption, cost, and area.

The VIRAM chip [5] is a PIM research prototype being developed at the University of California at Berkeley. A simplified architecture of the chip is shown in Figure 1. The VIRAM contains two vector-processing units in addition to a scalar-processing unit. These units are pipelined. The vector functional units can be partitioned into several smaller units, depending on the arithmetic precision required. For example, a vector functional unit can be partitioned into 4 units for 64-bit operations or 8 units for 32-bit operations. Some operations are allowed to execute on ALU0 only. It has 8K vector register file (32 registers).

It has 13 Mbytes of DRAM. There is a 256-bit data path between the processing units and DRAM. The DRAM is partitioned into two wings, each of which has four banks. It can access eight sequential 32-bit data elements per clock cycle. However, since there are four address generators, it can access only four strided 32-bit or 64-bit data elements per cycle.

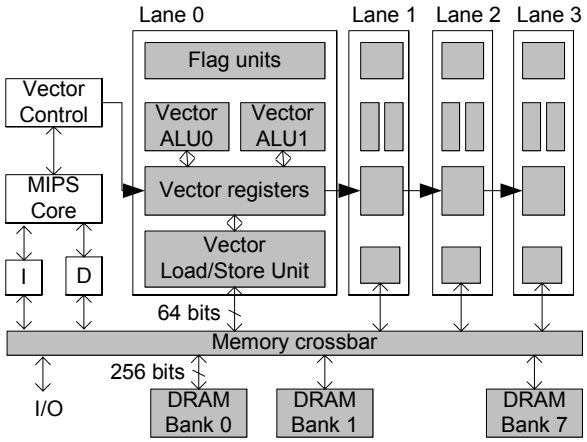


Figure 1. Block diagram of VIRAM

There is a crossbar switch between the DRAM and the vector processor. The target processor speed is 200 MHz, which would provide a peak performance of 3.2 GOPS (= 200 MHz x 2 ALUs x 8 data per clock) for 32-bit data. If 16-bit data is processed, the performance is 6.4 GOPS. Its peak floating point performance is 1.6 GFLOPS for 32-bit data. The power consumption is expected to be about 2 W. The EEMBC (Embedded Microprocessor Benchmark Consortium) benchmarks have been implemented on VIRAM [17]. VIRAM's performance is 20 times better (as measured by geometric mean normalized by clock frequency) than the K6-III+ x86 processor.

2.2 Imagine

Another approach for handling the growing processor-memory gap is stream processing. In this approach, the data is routed through stream registers to hide memory latency, allow the re-ordering of DRAM accesses, and minimize the number of memory accesses. The Imagine chip [4][11] is a research prototype stream processor developed at Stanford University. It contains eight clusters of arithmetic units that process data from a stream register file. The processor speed is currently 300 MHz, which provides a peak performance of over 14 GOPS (32-bit integer or floating-point operations). Performance results for Imagine have been presented for application kernels such as MPEG, and QRD [9]. ALU utilization between 84% and 95% is reported for streaming media applications.

Figure 2 shows the block diagram of Imagine. The stream processing is implemented with eight ALU clusters (with 6 ALUs each) with a large stream register file (SRF), and a high-bandwidth interconnect between them. The size of SRF is 128 Kbytes. A stream can start at the start of any SRF 128-byte blocks. Data is transferred to and from the SRF from off-chip memory

or the network interface. The eight ALU clusters operate on data from the SRF. Up to eight input or output streams can be processed simultaneously. The data is sent to clusters in round-robin fashion, i.e., the *i*-th data is sent to cluster (*i* mod 8). All clusters perform the same operations on their data in SIMD style. Each cluster has 6 arithmetic units (three adders, two multipliers, and one divider) and one communication interface that is used to send data between ALU clusters.

The Imagine prototype implementation has two memory controllers, each of which can process a memory access stream. The memory controller reorders accesses to reduce bank conflicts and to increase data access locality. The processor speed in the lab is currently near 300 MHz, which would provide a peak performance of 14.4 GFLOPS (= 300 MHz x 8 ALU clusters x 6 arithmetic units per cluster) for 32-bit data.

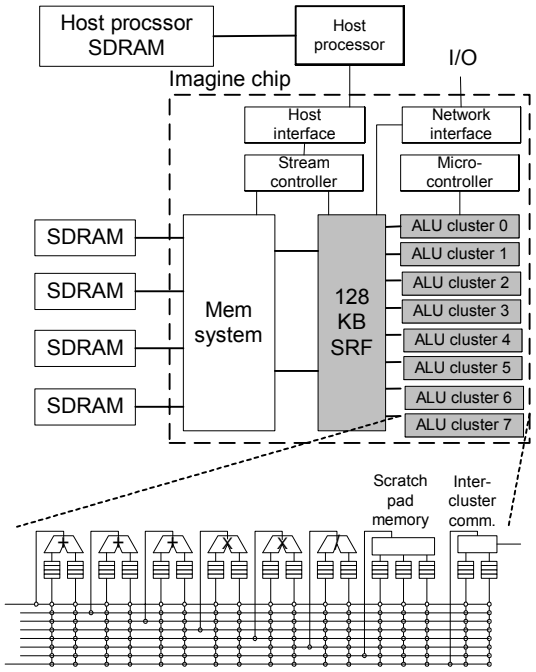


Figure 2. Block diagram of Imagine

2.3 Raw

Another approach for a scalable microprocessor that addresses issues of continued technology scaling is tile processing. Instead of building one processor on a chip, several processors (tiles) are implemented and connected in a mesh topology using a scalar operand network [16]. Then, each tile occupies a fraction of the chip space, so it is easier to make a faster processor since the signals need to travel only a short distance. One example is the Raw chip implemented at MIT [15] and shown in Figure 3. The current Raw implementation contains 16 tiles on a chip connected by a very low latency 2-D mesh network. The Raw prototype has been tested up to 199MHz and is expected to operate at 300MHz. Peak performance is 4.8 GOPS.

Each tile has a MIPS-based RISC processor with floating-point units and a total of 128 KB of SRAM, which includes switch instruction memory, tile (processor) instruction memory, and data memory. Raw uses general parallelism, which includes streaming, ILP, and data parallelism.

The Raw has four networks: two static networks and two dynamic networks. Communication on the static networks is performed by a switch processor in each tile [15]. The switch processor is located between the computation processor and the network and provides throughput to the tile processor of one word per cycle with a latency of three cycles between nearest neighbor tiles. One additional cycle of latency is added for each hop in the mesh through the static networks. When the dynamic network is used, data is sent to another tile in a packet. A packet contains header and data. If the data is smaller than a packet, dummy data is added to make a packet. If the data is larger than the packet, multiple packets are sent. The memory ports are located at the 16 peripheral ports of the chip. All tiles can access memory either through the dynamic network or through the static network.

Several kernels including matrix multiplication are implemented on Raw and the results are reported in [16]. The results show that Raw obtains speedup of up to 12 relative to single-tile performance on ILP benchmarks. Speedups greater than 16 can be achieved on streaming benchmarks when compared to a single-issue load/store RISC architecture because of a tile's ability to operate on data directly from the networks.

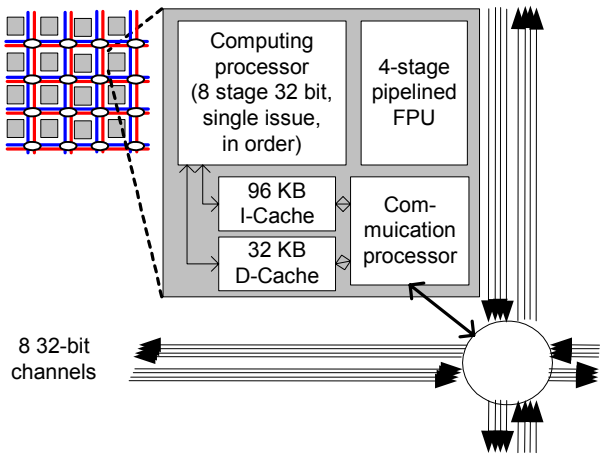


Figure 3. Block diagram of Raw

2.4 Programming methodologies

The programming methodologies and tools for each of these architectures are evolving. However, each architecture has inherent properties that affect the programming model and programmability of the architecture.

The VIRAM's programming model is that of a traditional vector architecture. An application is described as single instruction stream that contains scalar and vector instructions. There are two primary difficulties to programming the VIRAM architecture. First, the C programming language makes automatic parallelization of many loops difficult or impossible without making assumptions about the independence of pointer and array accesses. Simple loops or computations marked by user hints can be vectorized, but kernels with complex access patterns (e.g. FFT) are still difficult to automatically vectorize. Languages that are more restricted will facilitate automatic vectorization. The second factor that complicates the programmability of VIRAM is the impact of the DRAM organization on performance. Much of the performance of VIRAM is achieved by exploiting properties of DRAM organization (e.g. banks, rows, columns, and wings). Currently, the user must understand the DRAM organization to optimize performance. However, it is feasible that a compiler could organize memory references based on memory organization while it is vectorizing, especially given a language that makes this analysis feasible. For this study, a C compiler was used to compile the kernels, and then inner loops were hand-vectorized using assembly code.

The programming model of Imagine has two significant characteristics. First, the programming model is based on streams. Streams are similar to vectors, but streams can be explicitly routed between the stream register files and the ALU clusters without going through the memory system. This property is important for reducing the impact of the bandwidth bottleneck between DRAM and the processor chip. The second significant characteristic of the Imagine programming model is that a program is described in two languages, one for the host (or control) thread written in C and one for the stream processing unit written in kernel-C. Again, new programming languages may allow this distinction to be hidden from the programmer. However, the programming model used in this paper forces the programmer to think explicitly about streams and their control. This explicit streaming model has the disadvantage that a programmer must think about the application in a new way, but has the advantage that the programmer is forced to think about issues that are important to performance anyway. Applications must contain SIMD parallelism to see significant performance improvements on the Imagine architecture. For this study, inner loops were carefully scheduled to maximize performance.

The Raw architecture is the most flexible of the three architectures addressed in this paper. The tile-based organization with the low-latency, high-bandwidth network and memory interface supports a variety of programming models. The primary programming models used in the kernels described in this paper are the MIMD and stream models. The CSLC and beam steering kernels have plenty of independent parallelism to allow each tile to execute independently. We report results on two modes of using Raw: an easy-to-program but less efficient MIMD mode, in which data is routed to local memories through cache misses (CSLC), and a stream mode, in which data is routed in a stream mode without going through local memories by thinking explicitly about data placement and streams (beam steering).

However, the low-latency, high-bandwidth networks of Raw also allow ILP to be mapped efficiently to Raw. Raw's peak performance can be achieved when data can be operated on without going through local memories in the tiles. For this study, we used standard C to program the kernels. Assembly code was inserted only where necessary to access streaming data through the network. In beam steering, the codes between two instructions accessing streaming data through network are also written in assembly language. Other programming models, such as decoupled processing, are being developed for Raw and have the potential to improve performance of applications such as those described in this paper.

2.5 Performance models

In this section, simple performance models used to estimate the upper bound of the performance of the kernels on each architecture are described. We model computation and memory bandwidth. Memory latency is not modeled since these architectures can generally hide memory latency on the kernels used in this study.

Table 1 shows the DRAM memory and ALU throughput for 32-bit data elements that each architecture can support. It should be noted that both memory and ALU throughput are functions of these particular implementations and are not functions of the architectures themselves. However, the architectures provide the means to exploit the throughput supported by the implementation. It should also be noted that memory bandwidth reported is for the nearest DRAM. For VIRAM, DRAM is on-chip, while the nearest DRAM is off-chip for Imagine and Raw.

Table 1. Peak throughput (32-bit words per cycle)

	VIRAM	Imagine	Raw
On-chip DRAM Read/Write	8	16 (SRF)	16 (Cache)
Off-chip DRAM Read/Write	2 (Using DMA)	2	28
Computation	8	48	16

3. Kernel Implementations

In this section, three data-intensive kernels are described. Also, the techniques used to improve the performance on VIRAM, Imagine, and Raw are presented. The descriptions of the techniques are brief due to the space limitation.

3.1 Corner turn

The corner turn is a matrix transpose operation that tests memory bandwidth. The data in the source matrix is transposed and stored in the destination matrix. The matrix size used for this paper, which was chosen to be larger than Imagine's SRF (128 KB) and Raw's internal memories (2 MB), but smaller than VIRAM's on-chip memory (13 MB), is 1024 x 1024 with 4-byte elements.

Naive implementations of the corner turn can have poor performance because cache performance can be bad and strided data accesses degrade DRAM bandwidth. In conventional cache-based processor systems, tiling is used to reduce cache misses.

Our VRAM corner turn uses a blocking algorithm with a 16 x 16 element matrix. Blocking allows the vector registers to be used for temporary storage between the loads and stores. We used strided load operations with padding added to the matrix rows to avoid DRAM bank conflicts. Initial load latencies are not hidden. Stores are done sequentially from the vector registers to the memory.

On the Imagine processor, we divide the matrix into multi-row strips that allows us to use the stream register files. We use four input streams and one output stream simultaneously. Since the rows within a stream are read sequentially, we maximize memory bandwidth during the reading. The Imagine clusters are used to route data in the correct output order. The output data is transferred to memory in one stream. The output data is partitioned into 128 eight-word blocks. The eight words in a block are written sequentially, but the blocks are written with a non-unit stride.

Our corner turn on Raw uses one load and one store operation for each DRAM-to-DRAM transfer. The algorithm, designed at MIT and implemented at USC/ISI, was developed to ensure that all 16 Raw tiles are doing a load or store during as many cycles as possible and to avoid bottlenecks in the static networks and data ports. The algorithm operates on 64x64 word blocks that fit in a single local tile memory. Main memory operations are all done sequentially to maximize memory bandwidth since the transpose can be done in local memories, where all accesses are done in a single cycle.

3.2 Coherent side-lobe canceller (CSLC)

CSLC is a radar signal processing kernel used to cancel jammer signals caused by one or more jammers. Our CSLC implementation consists of FFTs, a weight application (multiplication) stage, and IFFTs. Most of the computation time is spent on the FFT and IFFT operations.

There are four input channels: two main channels and two auxiliary channels. Each channel has 8K samples per processing interval. All computations are done using single-precision floating-point operations. The data is partitioned into 73 overlapping sub-bands, each of which contains 128 samples, so 128-sample FFTs are used.

Since the majority of computation time on the CSLC is spent on the FFT operation, we improved the performance of the FFT by using the appropriate FFT algorithms for each architecture. In this study, a parallelized hand-optimized radix-4 FFT is used for VIRAM and Imagine. Note that since the size of the FFT for the CSLC is 128, which is not power of four, we used three radix-4 stages and one radix-2 stage. We did not hand-optimize our Raw FFT implementation. Rather, a C implementation of the radix-2 FFT is used for Raw because it provided better performance than the radix-4 FFT because of register spilling in the radix-4 FFT. The Raw implementation does independent data-parallel FFTs.

3.3 Beam steering

Beam steering is a radar-processing kernel that directs a phased-array radar without physically rotating the antenna. The computation of the phase for each antenna element stresses memory bandwidth and latency because large tables are used for calibration tables. Arithmetic operations are additions and shift operations.

In our implementation, the following parameters are used. The number of antenna elements is 1608. Each element can direct the signal up to 4 directions per dwell where a dwell is a period. The phase needs to be calculated for each direction using calibration data.

As for other kernels, we used hand-vectorization of the main portion of the beam steering on VIRAM. Since the same processing is performed for each data, the data is fed to the vector unit, which computes output data.

For the Imagine, a manually optimized kernel was written to maximize cluster ALU utilization. The input data streams are loaded into the stream register file and supplied to the clusters. The results are written back to memory through the register file.

The beam steering processing on each data is independent. Thus, on Raw, we partition the data among 16 tiles and each tile processes its own data. Input data is streamed through the static network and is operated on directly from the network.

4. Experimental results and analysis

4.1 Overview

In this section, the implementation results are presented. Performance of these kernels is obtained by using cycle-accurate simulators provided by the VIRAM, Imagine, and Raw teams.

For comparison purposes, actual measurements of performance were taken using a single node of a 1 GHz PowerPC G4-based system (Apple PowerMac G4) [1]. An implementation using AltiVec technology was used for speedup comparison. The Apple cc compiler was used with timing done using the MacOS X system call mach_absolute_time(). We manually inserted Altivec vector instructions.

Table 2 summarizes key parameters of each processor. Note that the PowerPC is a highly optimized chip in performance implemented with custom logic. However, other processors are research chips implemented using standard cells and very small design teams. Thus, if the same level of design effort were applied to these research architectures, we would expect much higher clock rates and density to be achieved.

In Table 3, a summary of the implementation results is shown. Figure 8 shows the speedup in terms of cycles and Figure 9 shows the speedup in terms of execution time. Note that Figure 8 and Figure 9 are both using a log scale on the vertical axis.

Table 2. Processor Parameters

	PPC G4	VIRAM	Imagine	Raw
Clock (MHz)	1000	200	300	300
# of ALUs	4	16	48	16
Peak GFLOPS	5	3.2	14.4	4.64

Table 3. Experimental results (cycles in 10^3)

	Corner Turn	CSLC	Beam Steering
PPC	34,250	29,013	730
Altivec	29,288	4,931	364
VIRAM	554	424	35
Imagine	1,439	196	87
Raw	146	357	19

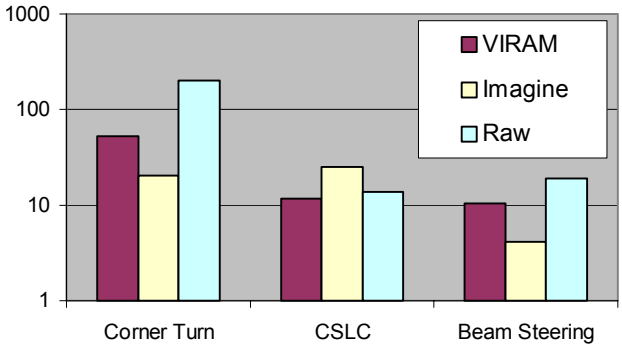


Figure 8. Speedup compared with PPC with AltiVec (Cycles)

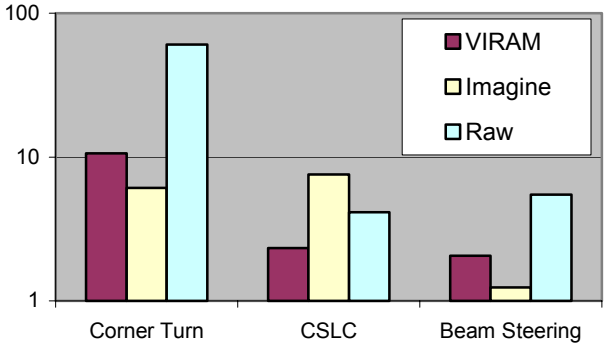


Figure 9. Speedup compared with PPC with AltiVec (execution times when PPC=1 GHz, VIRAM=200 MHz, Imagine=300 MHz, and Raw=300 MHz)

4.2 Corner turn

Table 4 summarizes the expected execution time using the performance model shown in Section 3. All three architectures provided speedups of more than 20 compared with a PowerPC system in terms of number of cycles. Corner turn performance is mostly a measure of memory bandwidth, which is not a direct property of an architecture, but rather a function of the number of pins in the package. However, the corner turn does demonstrate an architecture's ability to leverage memory bandwidth that does exist. Since VIRAM has on-chip DRAM, the kernel measures on-chip bandwidth. On the Imagine and Raw architectures, we're stressing off-chip memory.

The performance of corner turn on VIRAM is about half of what would have been expected from peak memory bandwidth. About 21% of the total cycles are overhead due to DRAM pre-charge cycles (which would be mostly hidden with sequential accesses) and TLB misses, and 24% are due to a limitation in strided load performance imposed by the number of address generators.

On Imagine, we assume the memory clock is the same frequency as the processor clock. Imagine has two address generators that provide two words per clock cycle. Note that the number of address generators is a processor implementation choice and is not a limitation of the stream architecture. Since the goal of the Imagine project was to demonstrate how memory traffic could be reduced, the Imagine team chose not to implement a high-bandwidth memory interface.

If network port were used to transfer data between SRF and an external memory connected to network port for corner turn, the performance would be the same since the network port has peak performance of two words per cycle.

87% of the cycles in the Imagine corner turn are due to memory transfers. The remaining 13% of the execution cycles are due to unoverlapped cluster instructions. Conceptually, the kernel instructions should be fully overlapped with memory accesses, but a limitation induced by the stream descriptor registers prevented full software pipelining in our implementation.

The Raw chip implementation actually provides enough main memory bandwidth that it is not the performance limiter for our corner turn implementation. Load/store issue rates and local memory bandwidth limit performance. 16 instructions per cycle are executed on the Raw tiles, and the static network and DRAM ports are not a bottleneck. The performance we achieved is nearly identical to the maximum performance predicted by the instruction issue rate. Memory latency is fully hidden (except for negligible start-up costs).

4.3 CSLC

CSLC mainly consists of FFTs and matrix-vector multiplication. Since the FFT length is 128, the working set fits into local memory, the performance of the CSLC depends primarily on ALU performance for Imagine and Raw.

Our IRAM CLSC analysis takes about 3.6 times longer than what is predicted by peak performance. The first factor reducing performance is overhead instructions. Instructions are needed to perform the FFT shuffles and increase the number of cycles by a factor of 1.67. The second factor that reduces FFT performance is ALU utilization. Since the second vector arithmetic unit in VIRAM cannot execute vector floating point instructions, performance on the FFT is reduced by a factor of 1.52. Finally, memory latency and vector startup costs increase performance by a factor of 1.41.

Imagine has the best performance of the three architectures on CSLC. This is because it is a computation-intensive kernel for which the working sets fit in the stream register files. Although the data access patterns for FFT are challenging for any architecture, the streaming execution model of Imagine is able to reduce memory operations and Imagine functions as intended on this kernel. Overall, performance achieved on CSLC on Imagine is about 20% of what is predicted by peak performance. While this is much lower than has been achieved for many media benchmark kernels, it still allows Imagine to perform about 10 useful operations per cycle; much better than can be achieved on today's superscalar architectures. Performance is reduced by 30% because inter-cluster communication is used to perform parallel FFTs. An alternative implementation, which was not completed for this study, would execute independent FFTs in parallel to eliminate inter-cluster communication overhead.

For the FFT kernel, ALU utilization (as measured by minimum FFT computations / total ALU cycles available) is 25.5%. If we exclude the divider, which is not useful for the FFT, then the utilization is 30.6%. Note that the utilization for the 128-point FFT is a little lower than the more than 40% obtained in other processing intensive applications [6]. The reason for the relatively low utilization is that the small size of the FFT reduces the amount of software pipelining and increases start-up overheads.

On Raw, we implemented a data parallel version of CSLC. The local memory on Raw successfully caches the working sets, and less than 10% of the execution time is spent on memory stalls. Note that most of this

stalling could have been eliminated by implementing a streaming DMA transfer to the local memory that is overlapped with the computation.

The CSLC on Raw uses radix-2 FFT to avoid register spilling encountered in the radix-4 FFT. The number of operations (including loads and stores) in the radix-2 FFT is about 1.5 the number in the radix-4 FFT. So care should be given when the performance of the Raw on CSLC is compared with CSLC performance on other architectures.

One problem with our data parallel implementation of CSLC on Raw was load balancing. The CSLC is easily parallelized for 16 tiles. However, since the number of data sets is 73, which is not a multiple of the number of tiles, some tiles processed five sets while others processed four sets. About 8% of CPU cycles are idle due to load balancing. However, the number of sets in a real environment is not fixed at 73. In a real implementation, the input data sets would arrive continuously. Therefore, it is reasonable to assume that Raw could have perfect load balancing in a real implementation. Thus, we report the performance numbers for CSLC on Raw based on an extrapolation that assumes perfect load balancing.

Raw achieves about 31.4% of the peak performance on CSLC. In addition to the cache stall time previously discussed, about 26% of the cycles on Raw are consumed by load and store instructions. The remaining cycles are consumed by address and index calculations and loop overhead instructions.

If FFT is implemented using the stream interface that uses static network, it hides the cache miss stalls, and load and store operations are not needed. A primitive implementation result suggests about 70% of FFT performance improvement.

4.4 Beam steering

Beam steering has small numbers of memory accesses (2 reads and 1 write) and computations (5 additions and 1 shift) per output data.

On VIRAM, the lower bound of the computation time is 56% of the simulation time. The difference between the expected time and simulation cycles (15,412) comes from waiting for the results from previous vector operations and the cycles needed to initialize the vector operations.

On Imagine, the computations and memory accesses for beam steering are overlapped. The performance is limited by memory bandwidth due to the relatively low number of computation per memory access. The load and store operations take 89% of the simulation time. The remaining 11% of execution time is due to the software pipeline prologue.

In an actual signal processing pipeline the beam steering kernel would stream its inputs from the proceeding kernel in the application (e.g., a poly-phase filter bank) and stream its outputs to the following kernel (e.g., per-beam equalization). In such a pipeline the performance of beam steering will not be limited by memory bandwidth, as in the case of this isolated kernel, but rather will be limited by arithmetic performance. On such a streaming application Imagine is expected to achieve a high fraction of its peak performance. If table values were read from the stream register file rather than memory on our kernel, performance would be increased by a factor of about two. The performance of a beam steering algorithm with more computation per data (which is a realistic assumption) could be much higher.

On Raw, we used the static network to stream data from memory while hiding memory latency. In this implementation, loads and stores are not necessary and ALU utilization is very high. The Raw beam steering implementation has the best performance of the three architectures because of the combination of memory bandwidth and high ALU utilization.

4.5 Altivec mapping

The PowerPC G4 provides a vector instruction set extension, which was used manually to achieve the G4 results shown in Section 4.1. The Altivec instruction set allows four 32-bit floating-point operations to be specified and executed in a single instruction. Using the AltiVec architecture gains a performance factor of about six for the CSLC and about two for beam steering and does not significantly improve performance for the corner turn, which is limited by main memory bandwidth.

4.6 Architecture comparison

VIRAM's primary advantage comes from the high bandwidth between the vector units and DRAM without paying the cost (in terms of pins and power) that are required to achieve high bandwidth between chips. VIRAM is especially suitable for vectorizable applications that can utilize the high bandwidth interface and that are small enough to fit in the on-chip memory. VIRAM outperformed the G4 Altivec by more than a factor of 10 on all three of our kernels and showed especially good performance on the kernels that emphasize memory bandwidth. For embedded applications with reasonably sized data sets, the VIRAM can be used as a one-chip system. If the application size is larger than the on-chip DRAM, the data needs to come

from off-chip memory and VIRAM would lose much of its advantage.

Imagine's high peak performance can be utilized in streaming applications where main memory accesses can be avoided or minimized. The CSLC kernel demonstrates that even when the Imagine ALUs are not fully utilized, performance can be quite high, especially when compared to a commercial microprocessor like the G4 Altivec. Imagine's stream-based architecture is designed for scalability and power efficiency and the Imagine architecture has the highest peak performance of the architectures in this study.

Raw also performs best on streaming applications since load and store operations can be eliminated and the static networks provide tremendous on-chip bandwidth. The kernels used in this study do not fully exploit this mode of execution. But we have shown that the tile structure of Raw can be used to utilize the memory bandwidth available from the external ports of Raw. The tile structure also provides flexible support for MIMD and ILP applications.

5. CONCLUSION

The authors have presented simulated performance results for three data-intensive radar processing kernels: the corner turn, coherent side-lobe canceller, and beam steering on systems based on three recent research processors (VIRAM, Imagine, and Raw). The results show that all three of these architectures have strengths and provide significant performance potential compared to the current generation of superscalar processors with vector extensions.

These emerging architectures demonstrate that they can be programmed quickly in high level languages and existing compilers to obtain adequate performance, while with hand optimization or future compilers, they can achieve performance that far outstrips existing architectures. Furthermore, all three of these architectures will scale as technology shrinks far better than today's superscalar processors.

6. ACKNOWLEDGMENTS

The authors gratefully acknowledge the extraordinary support of the UC Berkeley IRAM team, the Stanford Imagine team, and the MIT Raw team for the use of their compilers, simulators, and computational kernels and their generous help. This study obviously would not have been possible without their generous support.

The authors also appreciate comments, suggestions, and help from Krste Asanovic, Christos Kozyrakis, Bill Dally, Anant Agarwal, Brian Patrick Towles, Jung Ho Ahn, Abhishek Das, Brucek Khailany, Ujval J. Kapasi, John Owens, Michael.B.Taylor, Hank Hoffmann, Dong-In Kang, and Lavanya Swethranyan.

Effort sponsored by Defense Advanced Research Projects Agency (DARPA) through the Air Force Research Laboratory, USAF, under agreement number F30602-99-1-0521 and F30602-01-C-0171. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not interpreted as necessarily representing the official policies or endorsement, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), Air Force Research Laboratory, or the U.S. Government.

7. References

[1] Apple, http://www.apple.com/powermac/, 2002.

[2] M. Gordon, W Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe, "A Stream Compiler for Communication-Exposed Architectures, MIT Tech. Memo TM-627, Cambridge, MA, March, 2002.

[3] A. Gupta, J. L. Hennessy, K. Gharachorloo, T. Mowry, and W. D. Weber, "Computative Evaluation of Latency Reducing and Tolerating Techniques," Proc. 18th Annual International Symposium on Computer Architecture, Toronto, May 1991.

[4] B. Khailany, W. J. Dally, S. Rixner, U. J. Kapasi, P. Mattson, J. Namkoong, J. D. Owens, B. Towles, and A. Chang., "Imagine: Media Processing with Streams," IEEE Micro, March/April 2001, pp. 35-46.

[5] C. Kozyrakis, "Scalable Vector Media-processors for Embedded Systems," Ph. D. dissertation, UC Berkeley, May 2002.

[6] U. Kapasi, W. J. Dally, S. Rixner, J. D. Owens, and B. Khailany, "The Imagine Stream Processor," International Conference on Computer Design, Freiburg, Germany, September 2002.

[7] J. Hennessy and D. A. Patterson, Computer Architecture: A Quantitative Approach, 2nd Edition, Morgan Kaufmann Publishers, Inc., 1996.

[8] Mitsubishi Microcomputers, M32000D4BFP-80 Data Book, http://www.mitsubishichips.com/data/datasheets /mcus/ mcupdf/ds/e32r80.pdf.

[9] J. D. Owens, S. Rixner, U. J. Kapasi, P. Mattson, B. Towles, B. Serebrin, and W. J. Dally, "Media Processing Applications on the Imagine," Stream Processor Proceedings of International Conference on Computer Design, Freiburg, Germany, September 2002.

[10] S. A. Przybylski, Cache and Memory Hierarchy Design: A Performance-Directed Approach, Morgan Kaufmann Publishers, San Mateo, CA, 1990.

[11] S. Rixner, W. J. Dally, U. J. Kapasi, B. Khailany, A. Lopez-Lagunas, P. R. Mattson, and J. D. Owens, "A Bandwidth-Efficient Architecture for Media Processing," 31st Annual International Symposium on Microarchitecture, Dallas, Texas, November 1998.

[12] A. J. Smith, "Cache Memories," Computing Surveys, Vol. 14, No. 3, pp. 473-530, 1982.

[13] J. Suh and S.P. Crago, "PIM- and Stream Processor-based Processing for Radar Signal Applications," MSP 02, Austine, TX, 2002.

[14] J. Suh, S. P. Crago, C. Li, and R. Parker, "A PIM-based Multiprocessor System," International Parallel and Distributed Processing Symposium, San Francisco, CA, 2000.

[15] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffmann, P. Johnson, W. Lee, A. Saraf, N. Shnidman, V. Strumpen, S. Amarasinghe, and A. Agarwal, "A 16-issue multiple-program-counter microprocessor with point-to-point scalar operand network," *Proceedings of the IEEE International Solid-State Circuits Conference*, February 2003.

[16] M. B. Taylor, W. Lee, S. Amarasinghe, and A. Agarwal, "Scalar Operand Networks: On-chip Interconnect for ILP in Partitioned Architectures," International Symposium on High Performance Computer Architecture, February 2003.

[17] C. Kozyrakis, D. Patterson, "Vector Vs. Superscalar and VLIW Architectures for Embedded Multimedia Benchmarks," *35th International Symposium on Microarchitecture*, Instabul, Turkey, November 2002.